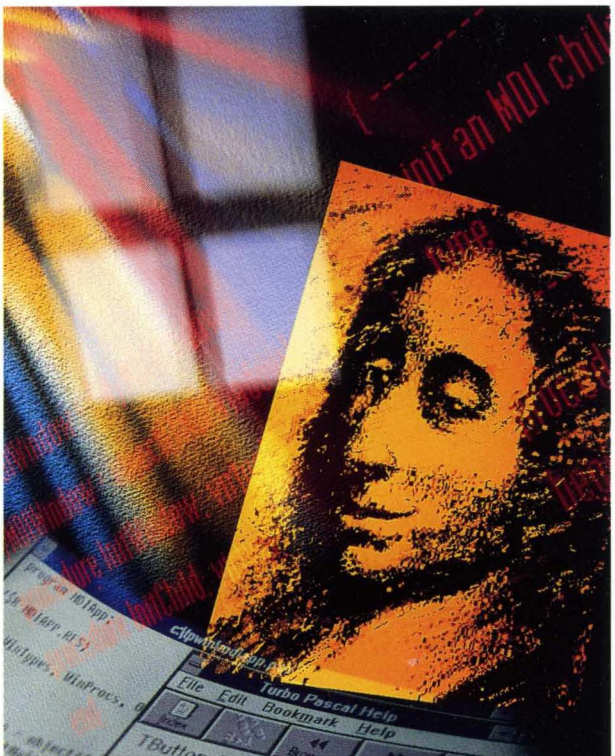
TURBO PASCAL®
FOR WINDOWS

PROGRAMMER'S GUIDE

BORLAND



Turbo Pascal® for Windows

Programmer's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

Copyright © 1987, 1991 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Windows, as used in this manual, shall refer to Microsoft's implementation of a windows system. Other brand and product names are trademarks or registered trademarks of their respective holders.

PRINTED IN THE USA.
R1 10 9 8 7

C O N T E N T S

Introduction 1
What's in this manual 1

Part 1 The Turbo Pascal standard

Chapter 1 Tokens and constants 5
Special symbols and reserved words 5
Identifiers 7
Labels 9
Numbers 9
Character strings 11
Constant declarations 12
Comments 13
Program lines 14

Chapter 2 Blocks, locality, and scope 15
Syntax 15
Rules of scope 17
Scope of interface and standard identifiers 18

Chapter 3 Types 21
Simple types 22
Ordinal types 22
Integer types 23
Boolean types 24
Char type 25
Enumerated types 25
Subrange types 26
Real types 27
Software floating point 27
80x87 floating point 27
String types 28
Structured types 28
Array types 29
Record types 30
Object types 33
Components and scope 35
Methods 36
Virtual methods 36
Dynamic methods 37
Instantiating objects 37
Set types 40
File types 40
Pointer types 41
Procedural types 42
Identical and compatible types 43
Type identity 43
Type compatibility 44
Assignment compatibility 44
The type declaration part 45

Chapter 4 Variables 47
Variable declarations 47
The data segment 48
The stack segment 48
Absolute variables 49
Variable references 50
Qualifiers 50
Arrays, strings, and indexes 51
Records and field designators 52
Object component designators 52
Pointers and dynamic variables 53
Variable typecasts 53

Chapter 5 Typed constants 57
Simple-type constants 58
String-type constants 59
Structured-type constants 59
Array-type constants 59
Record-type constants 61
Object-type constants 61
Set-type constants 62

Pointer-type constants 62
Procedural-type constants 63

Chapter 6 Expressions 65
Expression syntax 66
Operators 69
Arithmetic operators 69
Logical operators 70
Boolean operators 70
String operator 72
PChar operators 72
Set operators 73
Relational operators 73
Comparing simple types 74
Comparing strings 74
Comparing packed strings 74
Comparing pointers 74
Comparing character pointers 75
Comparing sets 75
Testing set membership 75
The @ operator 75
@ with a variable 76
@ with a value parameter 76
@ with a variable parameter 76
@ with a procedure or function 76
@ with a method 77
Function calls 77
Set constructors 78
Value typecasts 79
Procedural types in expressions 79

Chapter 7 Statements 83
Simple statements 83
Assignment statements 84
Object type assignments 84
Procedure statements 85
Method, constructor, and destructor calls 85
Goto statements 86
Structured statements 87
Compound statements 87
Conditional statements 88
If statements 88
Case statements 89
Repetitive statements 90
Repeat statements 90
While statements 91
For statements 92
With statements 94

Chapter 8 Procedures and functions 97
Procedure declarations 97
Near and far declarations 99
Export declarations 99
Interrupt declarations 100
Forward declarations 100
External declarations 101
Assembler declarations 102
Inline declarations 102
Function declarations 103
Method declarations 105
Constructors and destructors 106
Parameters 108
Value parameters 109
Variable parameters 109
Objects 110
Untyped variable parameters 110
Procedural types 111
Procedural variables 111
Procedural-type parameters 114

Chapter 9 Programs and units 117
Program syntax 117
The program heading 117
The uses clause 118
Unit syntax 118
The unit heading 119
The interface part 119
The implementation part 120
The initialization part 121
Indirect unit references 121
Circular unit references 122
Sharing other declarations 124

Chapter 10 Dynamic-link libraries 127
What is a DLL? 127
Using DLLs 128
Import units 129
Static and dynamic imports 131

Writing DLLs 132
The export procedure directive 133
The exports clause 133
Library initialization and exit code .. 135
Library programming notes 136
Global variables in a DLL 137
Global memory and files in a DLL ... 137
DLLs and the System unit 137
Run-time errors in DLLs 138
DLLs and stack segments 139

Part 2 The standard libraries

Chapter 11 The System unit 141
Standard procedures and functions 141
Flow control procedures 141
Dynamic allocation procedures ... 141
Dynamic allocation functions 142
Transfer functions 142
Arithmetic functions 143
Ordinal procedures 143
Ordinal functions 143
String procedures 143
String functions 144
Pointer and address functions 144
Miscellaneous functions 144
Miscellaneous procedures 145
File input and output 145
An introduction to file I/O 145
I/O functions 146
I/O procedures 146
Text files 147
Procedures 148
Functions 148
Untyped files 148
Procedures 149
The FileMode variable 149
Devices in Turbo Pascal 149
DOS devices 150
Text file devices 150
Predeclared variables 151
Uninitialized variables 151
Initialized variables 151

Chapter 12 The WinDos unit 153
Constants, types, and variables 153
Constants 153
Flag constants 153
File mode constants 154
File attribute constants 154
File name component string lengths 155
Return flags for FileSplit 155
Types 155
File record types 155
The TRegisters type 156
The TDateTime type 156
The TSearchRec type 156
Variables 157
The DosError variable 157
Procedures and functions 158
Date and time procedures 158
Interrupt support procedures 158
Disk status functions 158
File-handling procedures 158
File-handling functions 159
Directory-handling procedures ... 159
Directory-handling procedures ... 159
Environment-handling functions .. 159
Miscellaneous procedures 159
Miscellaneous functions 160

Chapter 13 The Strings unit 161
What is a null-terminated string? 161
Using null-terminated strings 162
Character pointers and string literals . 162
Character pointers and character arrays 163
Character pointer indexing 164
Character pointer operations 166
Null-terminated strings and standard procedures 167
Using the Strings unit 167

Chapter 14 The WinCrt unit 171
Using the WinCrt unit 171
Special characters 173
Line input 173

Variables 174
WindowOrg 174
WindowSize 174
ScreenSize 174
Cursor 175
Origin 175
InactiveTitle 175
AutoTracking 175
CheckEOF 175
CheckBreak 176
WindowTitle 176
Procedures and Functions 176
InitWinCrt 177
DoneWinCrt 177
WriteBuf 177
WriteChar 178
KeyPressed 178
ReadKey 178
ReadBuf 178
GotoXY 178
WhereX 179
WhereY 179
ClrScr 179
ClrEol 179
CursorTo 179
ScrollTo 179
TrackCursor 179
AssignCrt 180

Chapter 15 Using the 80x87 181
The 80x87 data types 183
Extended range arithmetic 183
Comparing reals 185
The 80x87 evaluation stack 185
Writing reals with the 80x87 186
Units using the 80x87 187
Detecting the 80x87 187
Emulation in assembly language 187

Part 3 Inside Turbo Pascal

Chapter 16 Memory issues 191
Code segments 191
Segment attributes 191
MOVEABLE or FIXED 191
PRELOAD or DEMANDLOAD ... 192
DISCARDABLE or PERMANENT . 192
Changing attributes 192
The automatic data segment 192
The heap manager 194
The HeapError variable 195
Internal data formats 197
Integer types 197
Char types 197
Boolean, WordBool, and LongBool types 197
Enumerated types 197
Floating-point types 198
The Real type 198
The Single type 198
The Double type 199
The Extended type 199
The Comp type 199
Pointer types 200
String types 200
Set types 200
Array types 200
Record types 201
File types 201
Procedural types 202
Direct memory access 202

Chapter 17 Objects 205
Internal data format of objects 205
Virtual method tables 206
Dynamic method tables 209
The SizeOf function 211
The TypeOf function 212
Virtual method calls 212
Dynamic method calls 213
Method calling conventions 214
Constructors and destructors 214
Extensions to New and Dispose 215
Assembly language methods 217
Constructor error recovery 220

Chapter 18 Control issues 225
Calling conventions 225
Variable parameters 226

Value parameters 226
Function results 227
NEAR and FAR calls 227
Nested procedures and functions ... 228
Entry and exit code 229
Register-saving conventions 231
Exit procedures 231
Interrupt handling 233
Writing interrupt procedures 233

Chapter 19 Input and output issues 235
Text file device drivers 235
The Open function 236
The InOut function 237
The Flush function 237
The Close function 237
Direct port access 237

Chapter 20 Automatic optimizations 239
Constant folding 239
Constant merging 240
Short-circuit evaluation 240
Order of evaluation 240
Range checking 241
Shift instead of multiply 241
Automatic word alignment 241
Dead code removal 242
Smart linking 242

Chapter 21 Compiler directives 245
Switch directives 246
Align data 247
Boolean evaluation 247
Debug information 248
Force far calls 249
Generate 80286 Code 249
Input/output checking 250
Local symbol information 250
Range checking 251
Stack-overflow checking 251
Var-string checking 252
Windows stack frames 252
Extended syntax 253
Parameter directives 253
Code segment attribute 254
Description 255
Include file 255
Link object file 255
Memory allocation sizes 256
Numeric coprocessor 256
Resource file 257
Conditional compilation 257
Conditional symbols 258
The DEFINE directive 260
The UNDEF directive 260
The IFDEF directive 260
The IFNDEF directive 261
The IFOPT directive 261
The ELSE directive 261
The ENDIF directive 261

Part 4 Using Turbo Pascal with assembly language

Chapter 22 The inline assembler 265
The **asm** statement 266
Register use 267
Assembler statement syntax 267
Labels 267
Prefix opcodes 269
Instruction opcodes 269
RET instruction sizing 270
Automatic jump sizing 270
Assembler directives 271
Operands 273
Expressions 274
Differences between Pascal and Assembler expressions 274
Expression elements 275
Constants 275
Numeric constants 275
String constants 276
Registers 277
Symbols 278
Expression classes 281
Expression types 282
Expression operators 284
Assembler procedures and functions .. 287

Chapter 23 Linking assembler code 291
Turbo Assembler and Turbo Pascal .. 292
Examples of assembly language routines 293
Turbo Assembler example 297
Inline machine code 298
Inline statements 298
Inline directives 300

Part 5 Library reference

Chapter 24 The run-time library 305
Sample procedure 305
Abs function 306
Addr function 306
Append procedure 307
ArcTan function 308
Assign procedure 308
AssignCrt procedure 309
BlockRead procedure 309
BlockWrite procedure 311
ChDir procedure 312
Chr function 312
Close procedure 313
ClrEol procedure 313
ClrScr procedure 313
Concat function 314
Copy function 314
Cos function 315
CreateDir procedure 315
CSeg function 315
CursorTo procedure 316
Dec procedure 316
Delete procedure 316
DiskFree function 317
DiskSize function 317
Dispose procedure 317
DoneWinCrt procedure 318
DosVersion function 318
DSeg function 319
Eof function (text files) 319
Eof function (typed, untyped files) 320
Eoln function 320
Erase procedure 321
Exit procedure 322
Exp function 322
FileExpand function 323
FilePos function 323
FileSearch function 324
FileSize function 325
FileSplit function 325
FillChar procedure 326
FindFirst procedure 327
FindNext procedure 328
Flush procedure 328
Frac function 329
FreeMem procedure 329
GetArgCount function 330
GetArgStr function 330
GetCBreak procedure 330
GetCurDir function 330
GetDate procedure 331
GetDir procedure 331
GetEnvVar function 332
GetFAttr procedure 332
GetFTime procedure 333
GetIntVec procedure 334
GetMem procedure 334
GetTime procedure 334
GetVerify procedure 335
GotoXY procedure 335
Halt procedure 335
Hi function 336
Inc procedure 336
InitWinCrt procedure 337
Insert procedure 337
Int function 337
Intr procedure 338
IOResult function 339
KeyPressed function 339
Length function 340
Ln function 340
Lo function 340
MaxAvail function 341
MemAvail function 342
MkDir procedure 342
Move procedure 343
MsDos procedure 343

New procedure 344
Odd function 344
Ofs function 345
Ord function 345
PackTime procedure 345
ParamCount function 346
ParamStr function 346
Pi function 347
Pos function 347
Pred function 347
Ptr function 348
Random function 348
Randomize procedure 349
ReadBuf function 349
Read procedure (text files) 349
Read procedure (typed files) 351
ReadKey function 351
Readln procedure 352
RemoveDir procedure 352
Rename procedure 352
Reset procedure 353
Rewrite procedure 354
RmDir procedure 355
Round function 355
RunError procedure 356
ScrollTo procedure 356
Seek procedure 356
SeekEof function 357
SeekEoln function 357
Seg function 358
SetCBreak procedure 358
SetCurDir procedure 358
SetDate procedure 359
SetFAttr procedure 359
SetFTime procedure 360
SetIntVec procedure 360
SetTextBuf procedure 360
SetTime procedure 362
SetVerify procedure 362
Sin function 362
SizeOf function 363
SPtr function 363
Sqr function 364
Sqrt function 364
SSeg function 364
Str procedure 364
StrCat function 365
StrComp function 366
StrCopy function 366
StrDispose function 367
StrECopy function 367
StrEnd function 368
StrIComp function 368
StrLCat function 368
StrLComp function 369
StrLCopy function 370
StrLen function 370
StrLIComp function 371
StrLower function 371
StrMove function 372
StrNew function 372
StrPas function 373
StrPCopy function 373
StrPos function 374
StrRScan function 375
StrScan function 375
StrUpper function 376
Succ function 376
Swap function 377
TrackCursor procedure 377
Trunc function 377
Truncate procedure 378
UnpackTime procedure 378
UpCase function 378
Val procedure 379
WhereX procedure 380
WhereY procedure 380
Write procedure (text files) 380
Write procedure (typed files) 382
WriteBuf function 383
WriteChar procedure 383
Writeln procedure 383

Part 6 Appendixes

Appendix A Error messages 387
Compiler error messages 387
Run-time errors 406

DOS errors 406
I/O errors 408
Fatal errors 409

Appendix B Reference materials 413
ASCII codes 413
Keyboard scan codes 416

Index 419

T A B L E S

1.1: Turbo Pascal reserved words 7
1.2: Turbo Pascal standard directives 7
3.1: Predefined integer types 23
3.2: Real data types 27
6.1: Precedence of operators 65
6.2: Binary arithmetic operations 69
6.3: Unary arithmetic operations 69
6.4: Logical operations 70
6.5: Boolean operations 71
6.6: String operation 72
6.7: Set operations 73
6.8: Relational operations 73
6.9: Pointer operation 75
13.1: Functions in the Strings unit 167
22.1: Values, classes, and types of symbols 279
22.2: Predefined type symbols 284
22.3: Inline asssembler expression operators 284
24.1: Components of the output string ..381
24.2: Components of the fixed-point string 382
B.1: ASCII table 414
B.2: Keyboard scan codes 417

F I G U R E S

16.1: Automatic data segment 193
17.1: Layouts of instances of TLocation, TPoint, and TCircle 206
17.2: TPoint and TCircle's VMT layouts . 208
17.3: TBase's VMT and DMT layouts 210
17.4: TDerived's VMT and DMT layouts . 211

INTRODUCTION

The User's Guide provides an overview of the entire Turbo Pascal documentation set. Read the introduction in that book for information on how to most effectively use the Turbo Pascal manuals.

This manual contains materials for the advanced programmer. If you already know how to program well (whether in Pascal or another language), this manual is for you. In it you will find a language reference; information on the standard libraries; programming information on memory and control issues, objects, floating point, dynamic-link libraries, assembly language interfacing, the run-time and compile-time error messages; and a complete reference to all of Turbo Pascal's procedures and functions.

Read the *User's Guide* if

1. You have never programmed in any language.
2. You have programmed, but not in Pascal, and you would like an introduction to the Pascal language.
3. You have programmed in Pascal but are not familiar with Borland's IDE (integrated development environment) for Windows.
4. You are looking for information on how to install Turbo Pascal.

The *User's Guide* also contains reference information on Turbo Pascal's IDE (including the editor), project managagement, and the command-line compiler.

What's in this manual

This book is split into six parts: language grammar, the standard libraries, advanced programming issues, interfacing with assembly language, definitions of the library routines, and run-time and compile-time error messages.

The first part of this manual, "The Turbo Pascal standard," offers technical information on the following features of the language:

- Chapter 1: "Tokens and constants"
- Chapter 2: "Blocks, locality, and scope"
- Chapter 3: "Types"
- Chapter 4: "Variables"
- Chapter 5: "Typed constants"
- Chapter 6: "Expressions"
- Chapter 7: "Statements"
- Chapter 8: "Procedures and functions"
- Chapter 9: "Programs and units"
- Chapter 10: "Dynamic-link libraries"

The second part contains information about all the standard libraries: the *System*, *WinDos*, *Strings*, *WinCrt*, *WinTypes*, and *WinProcs* units, along with information about using Turbo Pascal with a co-processor.

The third part provides further technical information for advanced users:

- Chapter 16: "Memory issues"
- Chapter 17: "Objects"
- Chapter 18: "Control issues"
- Chapter 19: "Input and output issues"
- Chapter 20: "Automatic optimizations"
- Chapter 21: "Compiler directives"

The fourth part discusses the issues involved with using Turbo Pascal with assembly language.

Part five contains all the definitions of the Turbo Pascal library routines, along with example program code to illustrate how to use these procedures and functions.

The two appendixes provide reference materials and list all the compiler and run-time error messages generated by Turbo Pascal.

PART 1

The Turbo Pascal standard

C H A P T E R

1

Tokens and constants

Tokens are the smallest meaningful units of text in a Pascal program, and they are categorized as special symbols, identifiers, labels, numbers, and string constants.

Separators cannot be part of tokens except in string constants.

A Pascal program is made up of tokens and separators, where a separator is either a blank or a comment. Two adjacent tokens must be separated by one or more separators if each token is a reserved word, an identifier, a label, or a number.

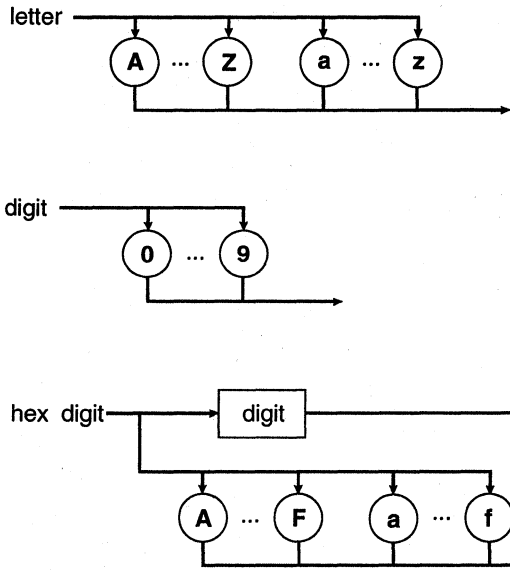
Special symbols and reserved words

Turbo Pascal uses the following subsets of the ASCII character set:

- **Letters**—the English alphabet, *A* through *Z* and *a* through *z*.
- **Digits**—the Arabic numerals 0 through 9.
- **Hex digits**—the Arabic numerals 0 through 9, the letters *A* through *F*, and the letters *a* through *f*.
- **Blanks**—the space character (ASCII 32) and all ASCII control characters (ASCII 0 through 31), including the end-of-line or return character (ASCII 13).

What follows are *syntax diagrams* for letter, digit, and hex digit. To read a syntax diagram, follow the arrows. Alternative paths are often possible; paths that begin at the left and end with an arrow on the right are valid. A path traverses boxes that hold the names of elements used to construct that portion of the syntax.

The names in rectangular boxes stand for actual constructions. Those in circular boxes—reserved words, operators, and punctuation—are the actual terms to be used in the program.



Special symbols and reserved words are characters that have one or more fixed meanings. These single characters are special symbols:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { } $ #
```

These character pairs are also special symbols:

```
<=  >=  :=  ..  (*  *)  (.  .)
```

Some special symbols are also operators. A left bracket (**[**) is equivalent to the character pair of left parenthesis and a period—**(.**. Similarly, a right bracket (**]**) is equivalent to the character pair of a period and a right parenthesis—**.)**.

Following are Turbo Pascal's reserved words:

Table 1.1
Turbo Pascal reserved words

and	**exports**	**nil**	**string**
asm	**file**	**not**	**then**
array	**for**	**object**	**to**
begin	**function**	**of**	**type**
case	**goto**	**or**	**unit**
const	**if**	**packed**	**until**
constructor	**implementation**	**procedure**	**uses**
destructor	**in**	**program**	**var**
div	**inline**	**record**	**while**
do	**interface**	**repeat**	**with**
downto	**label**	**set**	**xor**
else	**library**	**shl**	
end	**mod**	**shr**	

Reserved words appear in lowercase **boldface** throughout this manual. Turbo Pascal isn't case sensitive, however, so you can use either uppercase or lowercase letters in your programs.

The following are Turbo Pascal's standard directives. Unlike reserved words, these may be redefined by the user. However, this is not advised.

Table 1.2
Turbo Pascal standard directives

absolute	**external**	**index**	**near**
assembler	**far**	**interrupt**	**private**
export	**forward**	**name**	**resident**
			virtual

Note that **private** is a reserved word only within objects.

Identifiers

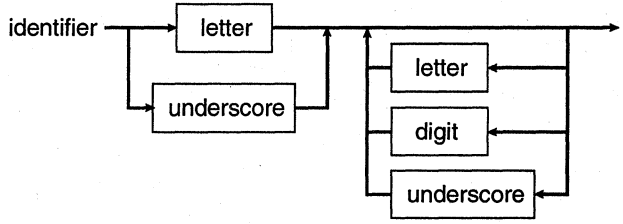
Identifiers denote constants, types, variables, procedures, functions, units, programs, and fields in records.

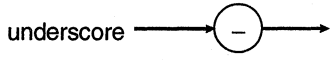
An identifier can be of any length, but only the first 63 characters are significant. An identifier must begin with a letter or an underscore character and cannot contain spaces. Letters, digits, and underscore characters (ASCII $5F) are allowed after the first character. Like reserved words, identifiers are not case sensitive.

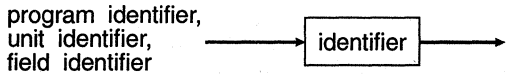
Units are described in Chapter 3 of the User's Guide and Chapter 9 of this manual.

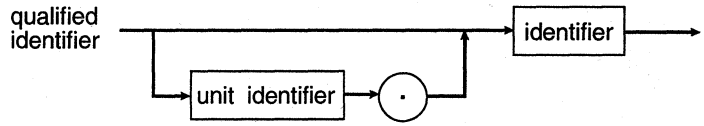
When several instances of the same identifier exist, you may need to qualify the identifier by a *unit identifier* in order to select a specific instance. For example, to qualify the identifier *Ident* by the

unit identifier *UnitName*, you would write *UnitName.Ident*. The combined identifier is called a *qualified identifier*.









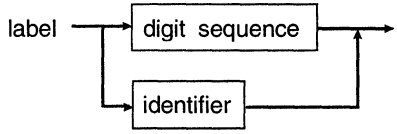
Here are some examples of identifiers:

```
Writeln
Exit
Real2String
System.MemAvail
Strings.StrLen
WinCrt.ReadText
```

In this manual, standard and user-defined identifiers are *italicized* when they are referred to in text.

Labels

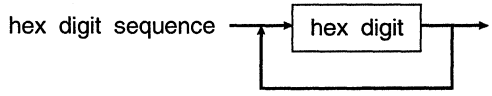
A *label* is a digit sequence in the range 0 to 9999. Leading zeros are not significant. Labels are used with **goto** statements.

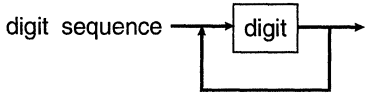


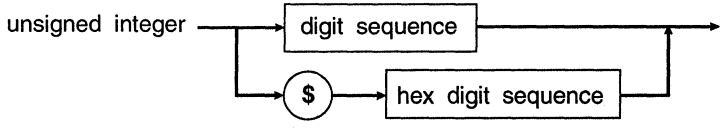
As an extension to standard Pascal, Turbo Pascal also allows identifiers to function as labels.

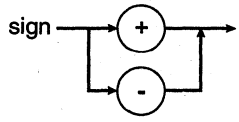
Numbers

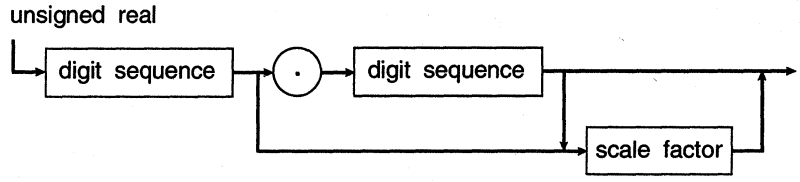
Ordinary decimal notation is used for numbers that are constants of type integer and real. A hexadecimal integer constant uses a dollar sign ($) as a prefix. Engineering notation (E or e, followed by an exponent) is read as "times ten to the power of" in real types. For example, 7E-2 means 7×10^{-2}; 12.25e+6 or 12.25e6 both mean $12.25 \times 10^{+6}$. Syntax diagrams for writing numbers follow:

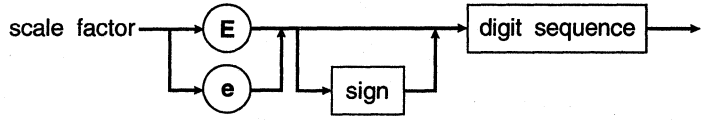


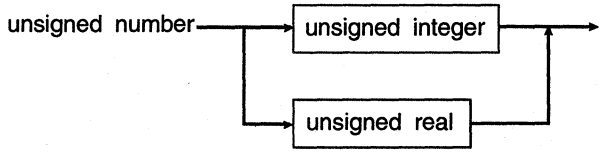


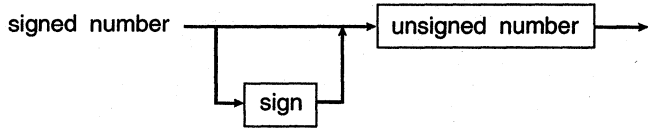












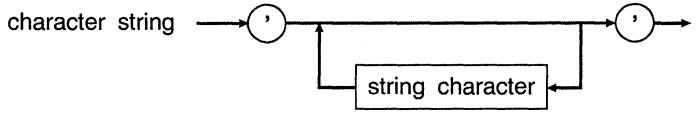
Numbers with decimals or exponents denote real-type constants. Other decimal numbers denote integer-type constants; they must be within the range –2,147,483,648 to 2,147,483,647.

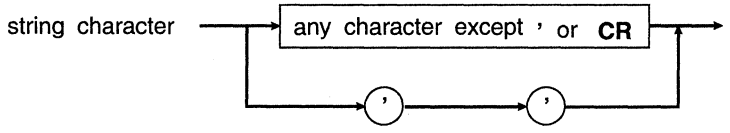
Hexadecimal numbers denote integer-type constants; they must be within the range $00000000 to $FFFFFFFF. The resulting value's sign is implied by the hexadecimal notation.

Character strings

A character string is a sequence of zero or more characters from the extended ASCII character set (Appendix B), written on one line in the program and enclosed by apostrophes. A character string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a character string denote a single character, an apostrophe. The length attribute of a character string is the actual number of characters within the apostrophes.

As an extension to standard Pascal, Turbo Pascal allows control characters to be embedded in character strings. The # character followed by an unsigned integer constant in the range 0 to 255 denotes a character of the corresponding ASCII value. There must be no separators between the # character and the integer constant. Likewise, if several control characters are part of a character string, there must be no separators between them.





A character string of length zero (the null string) is compatible only with string types. A character string of length one is compatible with any Char and string type. A character string of length *N*, where *N* is greater than or equal to 2, is compatible with any string type, with packed arrays of *N* characters, and with the PChar type when extended syntax is enabled {**$X+**}.

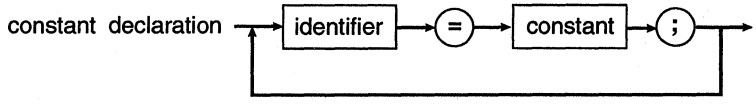
Here are some examples of character strings:

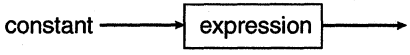
```
'TURBO'
'You'll see'
''''
';'
```

```
' '
#13#10
'Line 1'#13'Line2'
#7#7'Wake up!'#7#7
```

Constant declarations

A constant declaration declares an identifier that marks a constant within the block containing the declaration. A constant identifier cannot be included in its own declaration.





As an extension to standard Pascal, Turbo Pascal allows use of constant expressions. A *constant expression* is an expression that can be evaluated by the compiler without actually executing the program. Examples of constant expressions follow:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Turbo' + ' ' + 'Pascal'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Wherever standard Pascal allows only a simple constant, Turbo Pascal allows a constant expression.

The simplest case of a constant expression is a simple constant, such as 100 or 'A'.

Since the compiler has to be able to completely evaluate a constant expression at compile time, the following constructs are *not* allowed in constant expressions:

- references to variables and typed constants (except in constant address expressions, as described in Chapter 5).
- function calls (except those noted in the following text)

- the address operator (**@**) (except in constant address expressions, as described in Chapter 5)

For expression syntax, see Chapter 6, "Expressions."

Except for these restrictions, constant expressions follow the exact syntactical rules as ordinary expressions.

The following standard functions are allowed in constant expressions:

Abs	*Length*	*Ord*	*Round*	*Swap*
Chr	*Lo*	*Pred*	*SizeOf*	*Trunc*
Hi	*Odd*	*Ptr*	*Succ*	

Here are some examples of the use of constant expressions in constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrorStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Comments

The following constructs are comments and are ignored by the compiler:

```
{ Any text not containing right brace }
(* Any text not containing star/right parenthesis *)
```

The compiler directives are summarized in Chapter 21.

A comment that contains a dollar sign (**$**) immediately after the opening **{** or **(*** is a compiler directive. A mnemonic of the compiler command follows the **$** character.

Program lines

Turbo Pascal program lines have a maximum length of 126 characters.

C H A P T E R

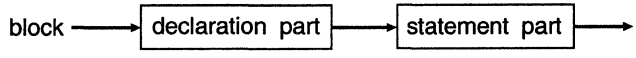
2

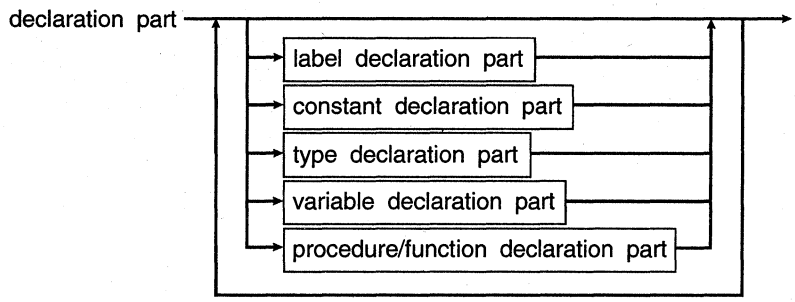
Blocks, locality, and scope

A block is made up of declarations, which are written and combined in any order, and statements. Each block is part of a procedure declaration, a function declaration, or a program or unit. All identifiers and labels declared in the declaration part are local to the block.

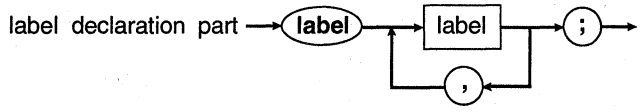
Syntax

The overall syntax of any block follows this format:



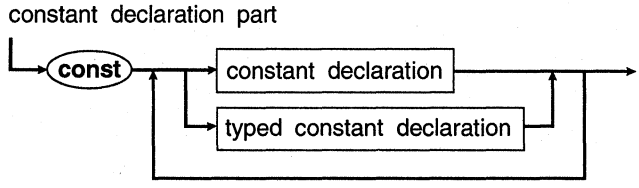


The *label declaration part* is where labels that mark statements in the corresponding statement part are declared. Each label must mark only one statement.

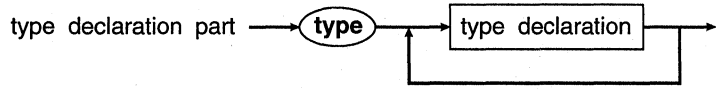


The digit sequence used for a label must be in the range 0 to 9999.

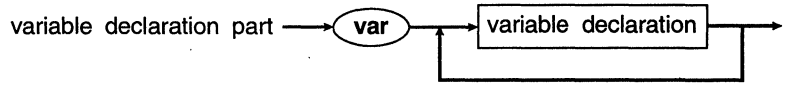
The *constant declaration part* consists of constant declarations local to the block.



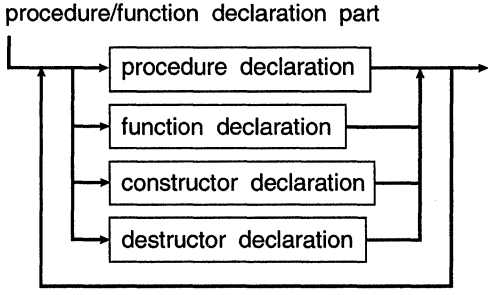
The *type declaration part* includes all type declarations local to the block.



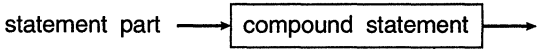
The *variable declaration part* is composed of variable declarations local to the block.



The *procedure and function declaration part* comprises procedure and function declarations local to the block.



The *statement part* defines the statements or algorithmic actions to be executed by the block.



Rules of scope

The presence of an identifier or label in a declaration defines the identifier or label. Each time the identifier or label occurs again, it must be within the *scope* of this declaration. The scope of an identifier or label encompasses its declaration to the end of the current block, including all blocks enclosed by the current block; some exceptions follow:

- **Redeclaration in an enclosed block:** Suppose that *Exterior* is a block that encloses another block, *Interior*. If *Exterior* and *Interior* both have an identifier with the same name (for example, *J*)

then *Interior* can only access the *J* it declared, and similarly *Exterior* can only access the *J* it declared.

- **Position of declaration within its block:** Identifiers and labels cannot be used until after they are declared. An identifier or label's declaration must come before any occurrence of that identifier or label in the program text, with one exception.
- The base type of a pointer type can be an identifier that has not yet been declared. However, the identifier must eventually be declared in the same type declaration part that the pointer type occurs in.
- **Redeclaration within a block:** An identifier or label can only be declared *once* in the outer level of a given block. The only exception to this is when it is declared within a contained block or is in a record's field list.
- A record field identifier is declared within a record type and is significant only in combination with a reference to a variable of that record type. So, you can redeclare a field identifier (with the same spelling) within the same block but not at the same level within the same record type. However, an identifier that has been declared can be redeclared as a field identifier in the same block.
- The scope of an object component's identifier extends over the domain of the object type. See page 35 for further explanation.

Scope of interface and standard identifiers

Programs or units containing **uses** clauses have access to the identifiers belonging to the interface parts of the units in those **uses** clauses.

Each unit in a **uses** clause imposes a new scope that encloses the remaining units used and the entire program. The first unit in a **uses** clause represents the outermost scope, and the last unit represents the innermost scope. This implies that if two or more units declare the same identifier, an unqualified reference to the identifier will select the instance declared by the last unit in the **uses** clause. However, by writing a qualified identifier, every instance of the identifier can be selected.

The identifiers of Turbo Pascal's predefined constants, types, variables, procedures, and functions act as if they were declared in a block enclosing all used units and the entire program. In fact,

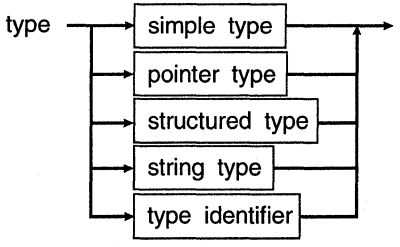
these standard objects are defined in a unit called *System*, which is used by any program or unit before the units named in the **uses** clause. This suggests that any unit or program can redeclare the standard identifiers, but a specific reference can still be made through a qualified identifier, for example, *System.Integer* or *System.Writeln*.

C H A P T E R

3

Types

When you declare a variable, you must state its type. A variable's *type* circumscribes the set of values it can have and the operations that can be performed on it. A *type declaration* specifies the identifier that denotes a type.



When an identifier occurs on the left side of a type declaration, it is declared as a type identifier for the block in which the type declaration occurs. A type identifier's scope does not include itself except for pointer types.

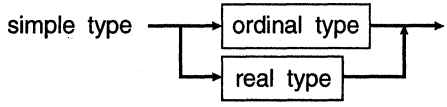
There are six major classes of types:

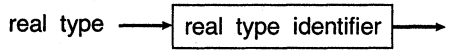
- simple types
- string types
- structured types
- pointer types
- procedural types
- object types

Each of these classes is described in the following sections.

Simple types

Simple types define ordered sets of values.





See Chapter 1 for how to denote constant type integer and real values.

A type real identifier is one of the standard identifiers: Real, Single, Double, Extended, or Comp.

Ordinal types

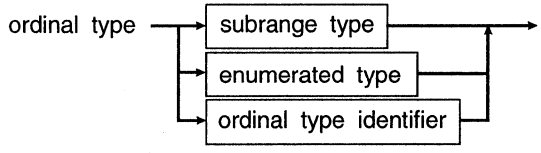
Ordinal types are a subset of simple types. All simple types other than real types are ordinal types, which are set off by four characteristics:

- All possible values of a given ordinal type are an ordered set, and each possible value is associated with an *ordinality*, which is an integral value. Except for type Integer values, the first value of every ordinal type has ordinality 0, the next has ordinality 1, and so on for each value in that ordinal type. A type Integer value's ordinality is the value itself. In any ordinal type, each

value other than the first has a predecessor, and each value other than the last has a successor based on the ordering of the type.

- The standard function *Ord* can be applied to any ordinal-type value to return the ordinality of the value.
- The standard function *Pred* can be applied to any ordinal-type value to return the predecessor of the value. If applied to the first value in the ordinal type, *Pred* produces an error.
- The standard function *Succ* can be applied to any ordinal-type value to return the successor of the value. If applied to the last value in the ordinal type, *Succ* produces an error.

The syntax of an ordinal type follows:



Turbo Pascal has nine predefined ordinal types: Integer, Shortint, Longint, Byte, Word, Boolean, WordBool, LongBool, and Char. In addition, there are two other classes of user-defined ordinal types: enumerated types and subrange types.

Integer types

There are five predefined integer types: Shortint, Integer, Longint, Byte, and Word. Each type denotes a specific subset of the whole numbers, according to the following table:

Table 3.1
Predefined integer types

Type	Range	Format
Shortint	–128 .. 127	Signed 8-bit
Integer	–32768 .. 32767	Signed 16-bit
Longint	–2147483648 .. 2147483647	Signed 32-bit
Byte	0 .. 255	Unsigned 8-bit
Word	0 .. 65535	Unsigned 16-bit

Arithmetic operations with type Integer operands use 8-bit, 16-bit, or 32-bit precision, according to the following rules:

- The type of an integer constant is the predefined integer type with the smallest range that includes the value of the integer constant.
- For a binary operator (an operator that takes two operands), both operands are converted to their common type before the operation. The *common type* is the predefined integer type with the smallest range that includes all possible values of both types. For instance, the common type of Integer and Byte is Integer, and the common type of Integer and Word is Longint. The operation is performed using the precision of the common type, and the result type is the common type.
- The expression on the right of an assignment statement is evaluated independently from the size or type of the variable on the left.
- Any byte-sized operand is converted to an intermediate word-sized operand that is compatible with both Integer and Word before any arithmetic operation is performed.

Typecasting is described in chapters 4 and 6.

An Integer type value can be explicitly converted to another integer type through typecasting.

Boolean types

There are three predefined boolean types: Boolean, WordBool, and LongBool. Boolean values are denoted by the predefined constant identifiers *False* and *True*. Because booleans are enumerated types, these relationships hold:

- False < True
- *Ord*(False) = 0
- *Ord*(True) = 1
- *Succ*(False) = True
- *Pred*(True) = False

A Boolean variable occupies one byte, a WordBool variable occupies two bytes (one word), and a LongBool variable occupies four bytes (two words). Boolean is the preferred type and uses the least memory; WordBool and LongBool primarily exist to provide compatibility with the Windows environment.

For reasons of compatibility with the Windows environment, booleans can assume ordinal values other than 0 and 1. A boolean expression is considered *False* when its ordinal value is zero, and

True when its ordinal value is non-zero. The **not**, **and**, **or**, and **xor** boolean operators work by testing for zero (*False*) or non-zero (*True*), but always return a result with an ordinal value of 0 or 1.

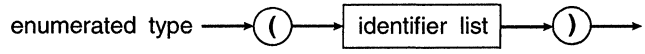
Char type

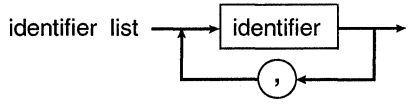
This type's set of values are characters, ordered according to the extended ASCII character set (Appendix B). The function call *Ord(Ch)*, where *Ch* is a Char value, returns *Ch*'s ordinality.

A string constant of length 1 can denote a constant character value. Any character value can be generated with the standard function *Chr*.

Enumerated types

Enumerated types define ordered sets of values by enumerating the identifiers that denote these values. Their ordering follows the sequence in which the identifiers are enumerated.





When an identifier occurs within the identifier list of an enumerated type, it is declared as a constant for the block in which the enumerated type is declared. This constant's type is the enumerated type being declared.

An enumerated constant's ordinality is determined by its position in the identifier list in which it is declared. The enumerated type in which it is declared becomes the constant's type. The first enumerated constant in a list has an ordinality of zero.

An example of an enumerated type follows:

```
type
  Suit = (Club, Diamond, Heart, Spade);
```

Given these declarations, *Diamond* is a constant of type *Suit*.

When the *Ord* function is applied to an enumerated type's value, *Ord* returns an integer that shows where the value falls with respect to the other values of the enumerated type. Given the

preceding declarations, *Ord(Club)* returns zero, *Ord(Diamond)* returns 1, and so on.

Subrange types

A subrange type is a range of values from an ordinal type called the *host type*. The definition of a subrange type specifies the smallest and the largest value in the subrange; its syntax follows:

Both constants must be of the same ordinal type. Subrange types of the form *A..B* require that *A* is less than or equal to *B*.

Examples of subrange types:

```
0..99
-128..127
Club..Heart
```

A variable of a subrange type has all the properties of variables of the host type, but its run-time value must be in the specified interval.

One syntactic ambiguity arises from allowing constant expressions where standard Pascal only allows simple constants. Consider the following declarations:

```
const
  X = 50;
  Y = 10;
type
  Color = (Red, Green, Blue);
  Scale = (X - Y) * 2..(X + Y) * 2;
```

Standard Pascal syntax dictates that, if a type definition starts with a parenthesis, it is an enumerated type, such as the *color* type described previously. However, the intent of the declaration of *scale* is to define a subrange type. The solution is to either reorganize the first subrange expression so that it does not start with a parenthesis, or to set another constant equal to the value of the expression, and then use that constant in the type definition:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real types

A real type has a set of values that is a subset of real numbers, which can be represented in floating-point notation with a fixed number of digits. A value's floating-point notation normally comprises three values—*M*, *B*, and *E*—such that $M \times B^E = N$, where *B* is always 2, and both *M* and *E* are integral values within the real type's range. These *M* and *E* values further prescribe the real type's range and precision.

There are five kinds of real types: Real, Single, Double, Extended, and Comp.

The real types differ in the range and precision of values they hold (see the next table).

Table 3.2
Real data types

Type	Range	Significant digits	Size in bytes
Real	2.9×10^{-39} .. 1.7×10^{38}	11-12	6
Single	1.5×10^{-45} .. 3.4×10^{38}	7-8	4
Double	5.0×10^{-324} .. 1.7×10^{308}	15-16	8
Extended	3.4×10^{-4932} .. 1.1×10^{4932}	19-20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19-20	8

The Comp type holds only integral values within the range $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18}.

Turbo Pascal supports two models of code generation for performing real-type operations: *software* floating point and *80x87* floating point. The appropriate model is selected through the **$N** compiler directive.

Software floating point

In the {**$N-**} state, which is selected by default, the code generated performs all real type calculations in software by calling run-time library routines. For reasons of speed and code size, only operations on variables of type real are allowed in this state. Any attempt to compile statements that operate on the Single, Double, Extended, and Comp types generates an error.

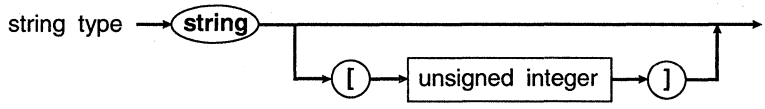
80x87 floating point

In the {**$N+**} state, the code generated performs all real type calculations using 80x87 instructions and can use all five real types.

String types

Type string operators are described in "String operator" and "Relational operators" in Chapter 6. Type string standard procedures and functions are described in "String procedures and functions" on page 143.

A type string value is a sequence of characters with a dynamic length attribute (depending on the actual character count during program execution) and a constant size attribute from 1 to 255. A string type declared without a size attribute is given the default size attribute 255. The length attribute's current value is returned by the standard function *Length*.



The ordering between any two string values is set by the ordering relationship of the character values in corresponding positions. In two strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a higher or greater-than value; for example, 'xs' is greater than 'x'. Null strings can only be equal to other null strings, and they hold the least string values.

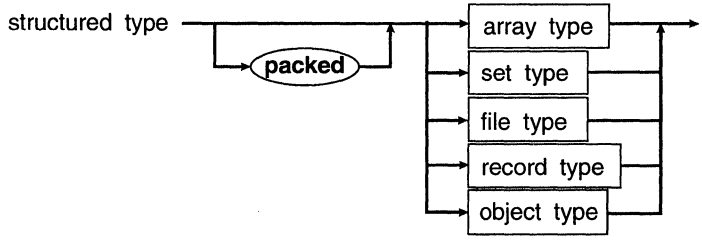
See the section "Arrays, strings, and indexes" in Chapter 4.

Characters in a string can be accessed as components of an array.

Structured types

The maximum permitted size of any structured type in Turbo Pascal is 65,520 bytes.

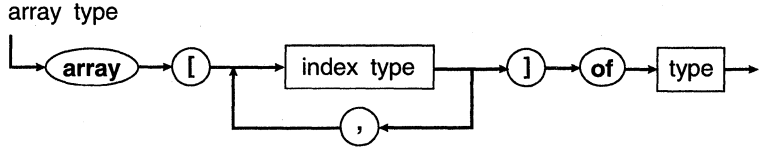
A structured type, characterized by its structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring.

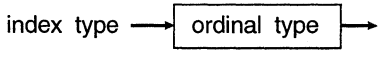


The word **packed** in a structured type's declaration tells the compiler to compress data storage, even at the cost of diminished access to a component of a variable of this type. The word **packed** has no effect in Turbo Pascal; instead packing occurs automatically whenever possible.

Array types

Arrays have a fixed number of components of one type—the component type. In the following syntax diagram, the component type follows the word **of**.





The index types, one for each dimension of the array, specify the number of elements. Valid index types are all ordinal types except Longint and subranges of Longint. The array can be indexed in each dimension by all values of the corresponding index type; the number of elements is therefore the number of values in each index type. The number of dimensions is unlimited.

The following is an example of an array type:

```
array[1..100] of Real
```

If an array type's component type is also an array, you can treat the result as an array of arrays or as a single multidimensional array. For instance,

```
array[Boolean] of array[1..10] of array[Size] of Real
```

is interpreted the same way by the compiler as

```
array[Boolean,1..10,Size] of Real
```

You can also express

```
packed array[1..10] of packed array[1..8] of Boolean
```

as

```
packed array[1..10,1..8] of Boolean
```

See "Arrays, strings, and indexes" in Chapter 4.

You access an array's components by supplying the array's identifier with one or more indexes in brackets.

An array type of the form

```
packed array[M..N] of Char
```

See "Identical and compatible types" later in this chapter.

where *M* is less than *N* is called a packed string type (the word **packed** can be omitted because it has no effect in Turbo Pascal). A packed string type has certain properties not shared by other array types.

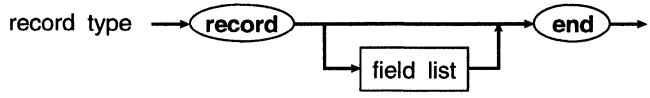
An array type of the form

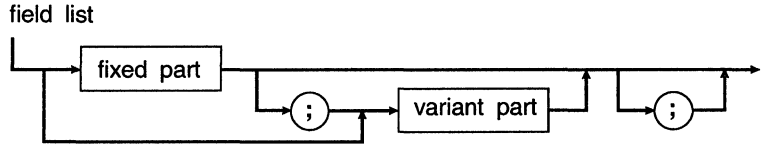
```
array[0..X] of Char
```

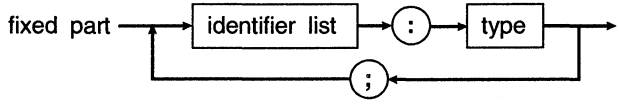
where *X* is a positive non-zero integer is called a zero-based character array. Zero-based character arrays are used to store *null-terminated strings*, and when the extended syntax is enabled (using a {**$X+**} compiler directive), a zero-based character array is compatible with a PChar value. For a complete discussion of this topic, refer to Chapter 13, "The Strings unit."

Record types

A record type comprises a set number of components, or fields, that can be of different types. The record type declaration specifies the type of each field and the identifier that names the field.





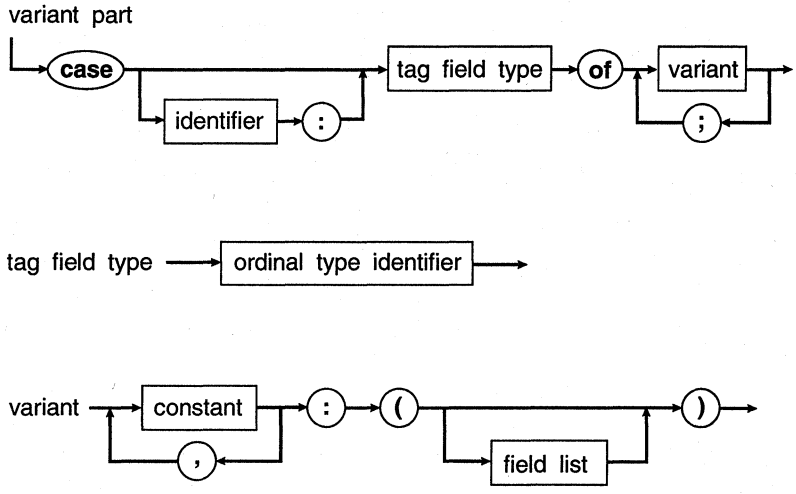


The fixed part of a record type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

The following is an example of a record type:

```
type
  DateRec = record
    Year: Integer;
    Month: 1..12;
    Day: 1..31;
  end;
```

The variant part shown in the syntax diagram of a record type declaration distributes memory space for more than one list of fields, so the information can be accessed in more ways than one. Each list of fields is a *variant*. The variants overlay the same space in memory, and all fields of all variants can be accessed at all times.



You can see from the diagram that each variant is identified by at least one constant. All constants must be distinct and of an ordinal type compatible with the tag field type. Variant and fixed fields are accessed the same way.

An optional identifier, the tag field identifier, can be placed in the variant part. If a tag field identifier is present, it becomes the identifier of an additional fixed field—the tag field—of the record. The program can use the tag field's value to show which variant is active at a given time. Without a tag field, the program selects a variant by another criterion.

Some record types with variants follow:

```
type
  Person = record
    FirstName, LastName: string[40];
    BirthDate: Date;
    case Citizen: Boolean of
      True: (BirthPlace: string[40]);
      False: (Country: string[20];
        EntryPort: string[20];
        EntryDate: Date;
        ExitDate: Date);
  end;

  Polygon = record
    X, Y: Real;
```

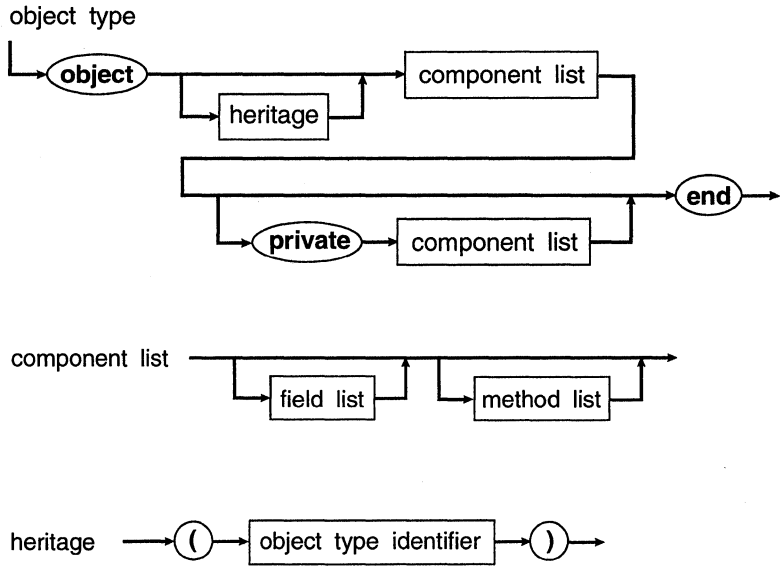
```
  case Kind: Figure of
    Rectangle: (Height, Width: Real);
    Triangle: (Size1, Side2, Angle: Real);
    Circle: (Radius: Real);
end;
```

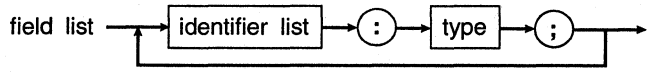
Object types

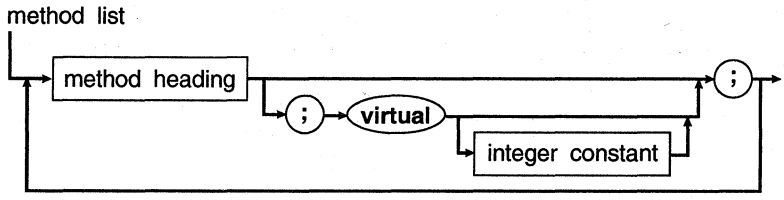
An object type is a structure consisting of a fixed number of components. Each component is either a *field*, which contains data of a particular type, or a *method*, which performs an operation on the object. Analogous to a variable declaration, the declaration of a field specifies the data type of the field and an identifier that names the field; and analogous to a procedure or function declaration, the declaration of a method specifies a procedure, function, constructor, or destructor heading.

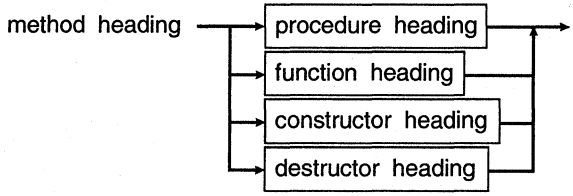
An object type can *inherit* components from another object type. If *T2* inherits from *T1*, then *T2* is a *descendant* of *T1*, and *T1* is an *ancestor* of *T2*.

Inheritance is transitive, that is, if *T3* inherits from *T2*, and *T2* inherits from *T1*, then *T3* also inherits from *T1*. The *domain* of an object type consists of itself and all its descendants.









The following code shows examples of object type declarations. These declarations are referred to by other examples throughout this chapter.

```
type
  Point = object
    X, Y: Integer;
  end;

  Rect = object
    A, B: Point;
    procedure Init(XA, YA, XB, YB: Integer);
    procedure Copy(var R: Rect);
    procedure Move(DX, DY: Integer);
    procedure Grow(DX, DY: Integer);
    procedure Intersect(var R: Rect);
    procedure Union(var R: Rect);
    function Contains(P: Point): Boolean;
  end;

  StringPtr = ^String;

  FieldPtr = ^Field;

  Field = object
    X, Y, Len: Integer;
```

```
  Name: StringPtr;
  constructor Copy(var F: Field);
  constructor Init(FX, FY, FLen: Integer; FName: String);
  destructor Done; virtual;
  procedure Display; virtual;
  procedure Edit; virtual;
  function GetStr: String; virtual;
  function PutStr(S: String): Boolean; virtual;
end;

StrFieldPtr = ^StrField;

StrField = object(Field)
  Value: StringPtr;
  constructor Init(FX, FY, FLen: Integer; FName: String);
  destructor Done; virtual;
  function GetStr: String; virtual;
  function PutStr(S: String): Boolean; virtual;
  function Get: String;
  procedure Put(S: String);
end;

NumFieldPtr = ^NumField;

NumField = object(Field)
  Value, Min, Max: Longint;
  constructor Init(FX, FY, FLen: Integer; FName: String;
    FMin, FMax: Longint);
  function GetStr: String; virtual;
  function PutStr(S: String): Boolean; virtual;
  function Get: Longint;
  procedure Put(N: Longint);
end;

ZipFieldPtr = ^ZipField;

ZipField = object(NumField)
  function GetStr: String; virtual;
  function PutStr(S: String): Boolean; virtual;
end;
```

Contrary to other types, an object type can be declared only in a type declaration part in the outermost scope of a program or unit. Thus, an object type cannot be declared in a variable declaration part or within a procedure, function, or method block.

Components and scope

The component type of a file type cannot be an object type, or any structured type with an object type component.

The scope of a component identifier extends over the domain of its object type. Furthermore, the scope of a component identifier

extends over procedure, function, constructor, and destructor blocks that implement methods of the object type and its descendants. For this reason, the spelling of a component identifier must be unique within an object type and all its descendants and all its methods.

The scope of a component identifier declared in the **private** section of an object type declaration is restricted to the module (program or unit) that contains the object type declaration. In other words, **private** component identifiers act like normal public component identifiers within the module that contains the object type declaration, but outside the module, any **private** component identifiers are unknown and inaccessible. By placing related object types in the same module, these object types can gain access to each others **private** components without making the **private** components known to other's modules.

Methods

The declaration of a method within an object type corresponds to a **forward** declaration of that method. Thus, somewhere after the object type declaration, and within the same scope as the object type declaration, the method must be *implemented* by a defining declaration.

When unique identification of a method is required, a *qualified method identifier* is used. It consists of an object type identifier, followed by a period (.), followed by a method identifier. Like any other identifier, a qualified method identifier can be prefixed with a unit identifier and a period if required.

Within an object type declaration, a method heading can specify parameters of the object type being declared, even though the declaration is not yet complete. This is illustrated by the *Copy*, *Intersect*, and *Union* methods of the *Rect* type in the previous example.

Virtual methods

Methods are by default *static*, but can, with the exception of constructor methods, be made *virtual* through the inclusion of a **virtual** directive in the method declaration. The compiler resolves calls to static methods at compile time, whereas calls to virtual methods are resolved at run time. The latter is sometimes referred to as *late binding*.

If an object type declares or inherits any virtual methods, then variables of that type must be *initialized* through a constructor call before any call to a virtual method. Thus, any object type that

declares or inherits any virtual methods must also declare or inherit at least one constructor method.

An object type can *override* (redefine) any of the methods it inherits from its ancestors. If a method declaration in a descendant specifies the same method identifier as a method declaration in an ancestor, then the declaration in the descendant overrides the declaration in the ancestor. The scope of an override method extends over the domain of the descendant in which it is introduced, or until the method identifier is again overridden.

An override of a static method is free to change the method heading in any way it pleases. In contrast, an override of a virtual method must match exactly the order, types, and names of the parameters, and the type of the function result, if any. Furthermore, the override must again include a **virtual** directive.

Dynamic methods

Turbo Pascal for Windows introduces an additional class of late-bound methods called *dynamic methods*. Dynamic methods differ from virtual methods only in the way dynamic method calls are dispatched at run time. For all other purposes, a dynamic method can be considered equivalent to a virtual method.

The declaration of a dynamic method is like that of a virtual method except that a dynamic method declaration must include a *dynamic method index* right after the **virtual** keyword. The dynamic method index must be an integer constant in the range 1..65535 and it must be unique among the dynamic method indices of any other dynamic methods contained in the object type or its ancestors. For example,

```
procedure FileOpen(var Msg: TMessage); virtual 100;
```

To learn more about dynamic methods and the differences in dispatching virtual and dynamic method calls, see Chapter 17, "Objects."

An override of a dynamic method must match the order, types, and names of the parameters and the type of the function result exactly. The override must also include a **virtual** directive followed by the same dynamic method index as was specified in the ancestor object type.

Instantiating objects

An object is *instantiated*, or created, through the declaration of a variable or typed constant of an object type, or by applying the *New* standard procedure to a pointer variable of an object type. The resulting object is called an *instance* of the object type.

```
var
  F: Field;
  Z: ZipField;
  FP: FieldPtr;
  ZP: ZipFieldPtr;
```

Given these variable declarations, *F* is an instance of *Field* and *Z* is an instance of *ZipField*. Likewise, after applying *New* to *FP* and *ZP*, *FP* points to an instance of *Field* and *ZP* points to an instance of *ZipField*.

If an object type contains virtual methods, then instances of that object type must be initialized through a constructor call before any call to a virtual method. Here's an example:

```
var
  S: StrField;
begin
  S.Init(1, 1, 25, 'Firstname');
  S.Put('Frank');
  S.Display;
  ...
  S.Done;
end;
```

If *S.Init* had not been called, then the call to *S.Display* would cause this example to fail.

Important! Assignment to an instance of an object type does *not* entail initialization of the instance.

The rule of required initialization also applies to instances that are components of structured types. For example,

```
var
  Comment: array[1..5] of StrField;
  I: Integer;
begin
  for I := 1 to 5 do Comment[I].Init(1, I + 10, 40, 'Comment');
  ...
  for I := 1 to 5 do Comment[I].Done;
end;
```

For dynamic instances, initialization is typically coupled with allocation, and cleanup is typically coupled with deallocation, using the extended syntax of the *New* and *Dispose* standard procedures. Here's an example:

```
var
  SP: StrFieldPtr;
```

```
begin
  New(SP, Init(1, 1, 25, 'Firstname'));
  SP^.Put('Frank');
  SP^.Display;
  ...
  Dispose(SP, Done);
end;
```

A pointer to an object type is assignment compatible with a pointer to any ancestor object type, therefore during execution of a program, a pointer to an object type might point to an instance of that type, or to an instance of any descendant type.

For example, a pointer of type *ZipFieldPtr* can be assigned to pointers of type *ZipFieldPtr*, *NumFieldPtr*, and *FieldPtr*, and during execution of a program, a pointer of type *FieldPtr* might be either **nil** or point to an instance of *Field*, *StrField*, *NumField*, or *ZipField*, or any other instance of a descendant of *Field*.

These pointer assignment compatibility rules also apply to object type variable parameters. For example, the *Field.Copy* method might be passed an instance of *Field*, *StrField*, *NumField*, *ZipField*, or any other instance of a descendant of *Field*.

A method is activated through a method designator of the form *Instance.Method*, where *Instance* is an instance of an object type, and *Method* is a method of that object type.

For static methods, the *declared* (compile-time) type of *Instance* determines which method to activate. For example, the designators *F.Init* and *FP^.Init* will always activate *Field.Init*, since the declared type of *F* and *FP^* is *Field*.

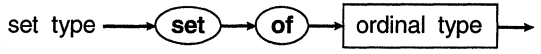
For virtual methods, the *actual* (run-time) type of *Instance* governs the selection. For example, the designator *FP^.Edit* might activate *Field.Edit*, *StrField.Edit*, *NumField.Edit*, or *ZipField.Edit*, depending on the actual type of the instance pointed to by *FP*.

In general, there is no way of determining which method will be activated by a virtual method designator. You can develop a routine (such as a forms editor input routine) that activates *FP^.Edit*, and later, without modifying that routine, apply it to an instance of a new, unforeseen descendant type of *Field*. When extensibility of this sort is desired, you should employ an object type with an open-ended set of descendant types, rather than a record type with a closed set of variants.

Set types

A set type's range of values is the power set of a particular ordinal type (the base type). Each possible value of a set type is a subset of the possible values of the base type.

A variable of a set type can hold from none to all the values of the set.



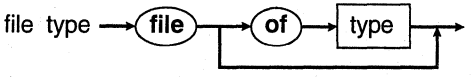
Set-type operators are described in the section entitled "Set operators" in Chapter 6. "Set constructors" in the same chapter shows how to construct set values.

The base type must not have more than 256 possible values, and the ordinal values of the upper and lower bounds of the base type must be within the range 0 to 255. For these reasons, the base type of a set cannot be Shortint, Integer, Longint, or Word.

Every set type can hold the value [ ], which is called the *empty set*.

File types

A file type consists of a linear sequence of components of the component type, which can be of any type except a file type or any structured type with a file-type component. The number of components is not set by the file-type declaration.

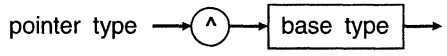


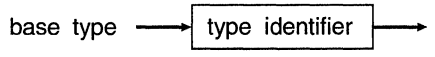
If the word **of** and the component type are omitted, the type denotes an untyped file. Untyped files are low-level I/O channels primarily used for direct access to any disk file regardless of its internal format.

The standard file type *text* signifies a file containing characters organized into lines. Text files use special input/output (I/O) procedures, which are discussed in Chapter 19, "Input and output issues."

Pointer types

A pointer type defines a set of values that point to dynamic variables of a specified type called the *base* type. A type Pointer variable contains the memory address of a dynamic variable.





If the base type is an undeclared identifier, it must be declared in the same type declaration part as the pointer type.

You can assign a value to a pointer variable with the *New* procedure , the @ operator, or the *Ptr* function. The *New* procedure allocates a new memory area in the application heap for a dynamic variable and stores the address of that area in the pointer variable. The @ operator directs the pointer variable to the memory area containing any existing variable, including variables that already have identifiers. The *Ptr* function points the pointer variable to a specific memory address.

The reserved word **nil** denotes a pointer-valued constant that does not point to anything.

See Chapter 4's section entitled "Pointers and dynamic variables" for the syntax of referencing the dynamic variable pointed to by a pointer variable.

The predefined type Pointer denotes an untyped pointer, that is, a pointer that does not point to any specific type. Variables of type Pointer cannot be dereferenced; writing the pointer symbol ^ after such a variable is an error. Like the value denoted by the word **nil**, values of type Pointer are compatible with all other pointer types.

The predefined type PChar denotes a pointer to a null-terminated string. PChar is declared as

```
type PChar = ^Char;
```

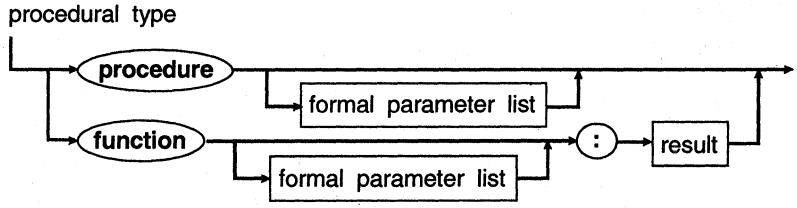
Turbo Pascal for Windows supports a set of *extended syntax* rules to facilitate handling of null-terminated strings using the PChar type. For a complete discussion of this topic, please refer to Chapter 13, "The Strings unit."

Procedural types

For a complete discussion of procedural types, refer to the "Procedural types" section on page 111.

Standard Pascal regards procedures and functions strictly as program parts that can be executed through procedure or function calls. Turbo Pascal has a much broader view of procedures and functions: It allows procedures and functions to be treated as objects that can be assigned to variables and passed as parameters. Such actions are made possible through *procedural types*.

A procedural type declaration specifies the parameters and, for a function, the result type.



In essence, the syntax for writing a procedural type declaration is exactly the same as for writing a procedure or function header, except that the identifier after the **procedure** or **function** keyword is omitted. Some examples of procedural type declarations follow:

```
type
  Proc = procedure;
  SwapProc = procedure(var X, Y: Integer);
  StrProc = procedure(S: string);
  MathFunc = function(X: Real): Real;
  DeviceFunc = function(var F: text): Integer;
  MaxFunc = function(A, B: Real; F: MathFunc): Real;
```

The parameter names in a procedural type declaration are purely decorative—they have no effect on the meaning of the declaration.

Turbo Pascal does not let you declare functions that return procedural type values; a function result value must be a string, Real, Integer, Char, Boolean, Pointer, or a user-defined enumeration.

Identical and compatible types

Two types may be the same, and this sameness (identity) is mandatory in some contexts. At other times, the two types need only be compatible or merely assignment-compatible. They are identical when they are declared with, or their definitions stem from, the same type identifier.

Type identity

Type identity is required only between actual and formal variable parameters in procedure and function calls.

Two types—say, *T1* and *T2*—are identical if one of the following is True: *T1* and *T2* are the same type identifier; *T1* is declared to be equivalent to a type identical to *T2*.

The second condition connotes that *T1* does not have to be declared directly to be equivalent to *T2*. The type declarations

```
T1 = Integer;
T2 = T1;
T3 = Integer;
T4 = T2;
```

result in *T1*, *T2*, *T3*, *T4*, and Integer as identical types. The type declarations

```
T5 = set of Integer;
T6 = set of Integer;
```

don't make *T5* and *T6* identical, since **set of** Integer is not a type identifier. Two variables declared in the same declaration, for example,

```
V1, V2: set of Integer;
```

are of identical types—unless the declarations are separate. The declarations

```
V1: set of Integer;
V2: set of Integer;
V3: Integer;
V4: Integer;
```

mean *V3* and *V4* are of identical type, but not *V1* and *V2*.

Type compatibility

Compatibility between two types is sometimes required, such as in expressions or in relational operations. Type compatibility is important, however, as a precondition of assignment compatibility.

Type compatibility exists when at least one of the following conditions is True:

- Both types are the same.
- Both types are real types.
- Both types are integer types.
- One type is a subrange of the other.
- Both types are subranges of the same host type.
- Both types are set types with compatible base types.
- Both types are packed string types with an identical number of components.
- One type is a string type and the other is a string type, packed string type, or Char type.
- One type is Pointer and the other is any pointer type.
- One type is PChar and the other is a zero-based character array of the form **array**[0..X] **of** Char. (This applies only when extended syntax is enabled with the {**$X+**} directive.)
- Both types are procedural types with identical result types, an identical number of parameters, and a one-to-one identity between parameter types.

Assignment compatibility

Assignment compatibility is necessary when a value is assigned to something, such as in an assignment statement or in passing value parameters.

A value of type T_2 is assignment-compatible with a type T_1 (that is, $T_1 := T_2$ is allowed) if any of the following are True:

- T_1 and T_2 are identical types and neither is a file type or a structured type that contains a file-type component at any level of structuring.
- T_1 and T_2 are compatible ordinal types, and the values of type T_2 falls within the range of possible values of T_1.

- T_1 and T_2 are real types, and the value of type T_2 falls within the range of possible values of T_1.
- T_1 is a real type, and T_2 is an integer type.
- T_1 and T_2 are string types.
- T_1 is a string type, and T_2 is a Char type.
- T_1 is a string type, and T_2 is a packed string type.
- T_1 and T_2 are compatible, packed string types.
- T_1 and T_2 are compatible set types, and all the members of the value of type T_2 fall within the range of possible values of T_1.
- T_1 and T_2 are compatible pointer types.
- T_1 is a PChar and T_2 is a string constant. (This applies only when extended syntax is enabled {**$X+**}.)
- T_1 is a PChar and T_2 is a zero-based character array of the form **array**[0..X] **of** Char. (This applies only when extended syntax is enabled {**$X+**}.)
- T_1 and T_2 are compatible procedural types.
- T_1 is a procedural type, and T_2 is a procedure or function with an identical result type, an identical number of parameters, and a one-to-one identity between parameter types.
- An object type T_2 is assignment compatible with an object type T_1 if T_2 is in the domain of T_1.
- A pointer type P_2, pointing to an object type T_2, is assignment compatible with a pointer type P_1, pointing to an object type T_1, if T_2 is in the domain of T_1.

A compile or run-time error occurs when assignment compatibility is necessary and none of the items in the preceding list are True.

The type declaration part

Programs, procedures, and functions that declare types have a type declaration part. An example of this follows:

```
type
  Range = Integer;
  Number = Integer;
  Color = (Red, Green, Blue);
  CharVal = Ord('A')..Ord('Z');
  TestIndex = 1..100;
```

```
TestValue = -99..99;
TestList = array[TestIndex] of TestValue;
TestListPtr = ^TestList;
Date = object
  Year: Integer;
  Month: 1..12;
  Day: 1..31;
  procedure SetDate(D, M, Y: Integer);
  function ShowDate: String;
end;
MeasureData = record
  When: Date;
  Count: TestIndex;
  Data: TestListPtr;
end;
MeasureList = array[1..50] of MeasureData;
Name = string[80];
Sex = (Male, Female);
Person = ^PersonData;
PersonData = record
  Name, FirstName: Name;
  Age: Integer;
  Married: Boolean;
  Father, Child, Sibling: Person;
  case S: Sex of
    Male: (Bearded: Boolean);
    Female: (Pregnant: Boolean);
end;
PersonBuf = array[0..SizeOf(PersonData)-1] of Byte;
People = file of PersonData;
```

In the example, *Range, Number,* and Integer are identical types. *TestIndex* is compatible and assignment-compatible with, but not identical to, the types *Number, Range,* and Integer. Notice the use of constant expressions in the declarations of *CharVal* and *PersonBuf.*

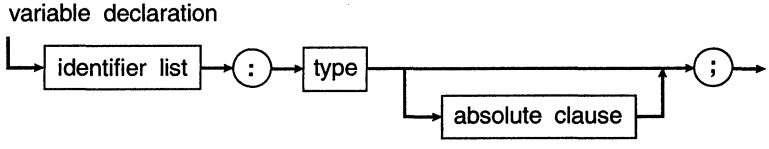
C H A P T E R

4

Variables

Variable declarations

A variable declaration embodies a list of identifiers that designate new variables and their type.



The type given for the variable(s) can be a type identifier previously declared in a **type** declaration part in the same block, in an enclosing block, or in a unit; it can also be a new type definition.

When an identifier is specified within the identifier list of a variable declaration, that identifier is a variable identifier for the block in which the declaration occurs. The variable can then be referred to throughout the block, unless the identifier is redeclared in an enclosed block. Redeclaration causes a new variable using the same identifier, without affecting the value of the original variable.

An example of a variable declaration part follows:

```
var
  X, Y, Z: Real;
  I, J, K: Integer;
  Digit: 0..9;
  C: Color;
  Done,Error: Boolean;
  Operator: (Plus, Minus, Times);
  Hue1, Hue2: set of Color;
  Today: Date;
  Results: MeasureList;
  P1, P2: Person;
  Matrix: array[1..10, 1..10] of Real;
```

Variables declared outside procedures and functions are called *global variables*, and reside in the *data segment*. Variables declared within procedures and functions are called *local variables*, and reside in the *stack segment*.

The data segment

The maximum size of the data segment is 65,520 bytes. When a program is linked (this happens automatically at the end of the compilation of a program), the global variables of all units used by the program, as well as the program's own global variables, are placed in the data segment.

For further details on this subject, see "Pointers and dynamic variables" on page 53.

If you need more than 65,520 bytes of global data, you should allocate the larger structures as dynamic variables.

The stack segment

The size of the stack segment is set through a **$M** compiler directive—it can be anywhere from 1,024 to 65,520 bytes. The default stack segment size is 8192 bytes.

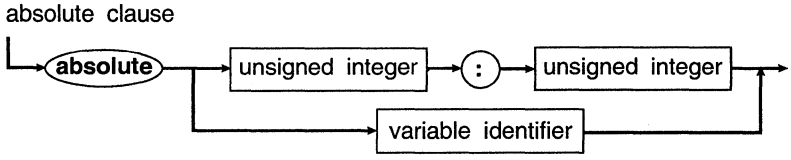
Each time a procedure or function is activated (called), it allocates a set of local variables on the stack. On exit, the local variables are disposed. At any time during the execution of a program, the total size of the local variables allocated by the active procedures and functions cannot exceed the size of the stack segment.

The **$S** compiler directive is used to include stack overflow checks in the code. In the default {**$S+**} state, code is generated to check for stack overflow at the beginning of each procedure and function. In the {**$S-**} state, no such checks are performed. A stack

overflow may very well cause a system crash, so don't turn off stack checks unless you are absolutely sure that an overflow will never occur.

Absolute variables

Variables can be declared to reside at specific memory addresses, and are then called *absolute variables*. The declaration of such variables must include an **absolute** clause following the type:



Note that the variable declaration's identifier list can only specify one identifier when an **absolute** clause is present.

The first form of the **absolute** clause specifies the segment and offset at which the variable is to reside:

```
CrtMode : Byte absolute $0040:$0049;
```

The first constant specifies the segment base, and the second specifies the offset within that segment. Both constants must be within the range $0000 to $FFFF (0 to 65,535).

The second form of the **absolute** clause is used to declare a variable "on top" of another variable, meaning it declares a variable that resides at the same memory address as another variable.

```
var
  Str    : string[32];
  StrLen : Byte absolute Str;
```

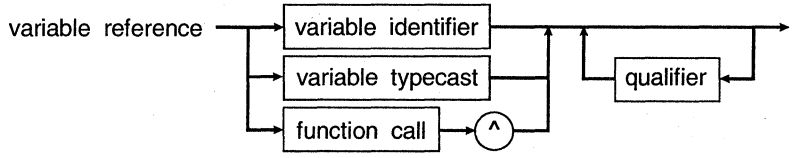
This declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and because the first byte of a string variable contains the dynamic length of the string, *StrLen* will contain the length of *Str*.

Variable references

A variable reference signifies one of the following:

- a variable
- a component of a structured- or string-type variable
- a dynamic variable pointed to by a pointer-type variable

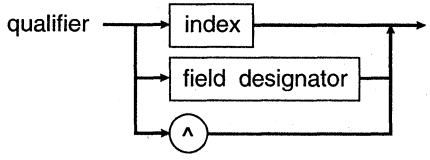
The syntax for a variable reference is



Note that the syntax for a variable reference allows a function call to a pointer function. The resulting pointer is then dereferenced to denote a dynamic variable.

Qualifiers

A variable reference is a variable identifier with zero or more qualifiers that modify the meaning of the variable reference.



An array identifier with no qualifier, for example, references the entire array:

```
Results
```

An array identifier followed by an index denotes a specific component of the array—in this case a structured variable:

```
Results[Current + 1]
```

With a component that is a record or object, the index can be followed by a field designator; here the variable access signifies a specific field within a specific array component.

```
Results[Current + 1].Data
```

The field designator in a pointer field can be followed by the pointer symbol (a ^) to differentiate between the pointer field and the dynamic variable it points to.

```
Results[Current + 1].Data^
```

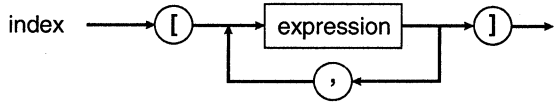
If the variable being pointed to is an array, indexes can be added to denote components of this array.

```
Results[Current + 1].Data^[J]
```

Arrays, strings, and indexes

A specific component of an array variable is denoted by a variable reference that refers to the array variable, followed by an index that specifies the component.

A specific character within a string variable is denoted by a variable reference that refers to the string variable, followed by an index that specifies the character position.



The index expressions select components in each corresponding dimension of the array. The number of expressions can't exceed the number of index types in the array declaration. Furthermore, each expression's type must be assignment-compatible with the corresponding index type.

When indexing a multidimensional array, multiple indexes or multiple expressions within an index can be used interchangeably. For example,

```
Matrix[I][J]
```

is the same as

```
Matrix[I, J]
```

You can index a string variable with a single index expression, whose value must be in the range 0..*N*, where *N* is the declared size of the string. This accesses one character of the string value, with the type *Char* given to that character value.

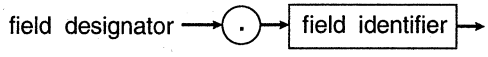
The first character of a string variable (at index 0) contains the dynamic length of the string; that is, *Length(S)* is the same as *Ord(S[0])*. If a value is assigned to the length attribute, the compiler does not check whether this value is less than the declared size of the string. It is possible to index a string beyond its current dynamic length. The characters thus read are random, and assignments beyond the current length will not affect the actual value of the string variable.

For more about indexing PChars, see Chapter 13, "The Strings unit."

When the extended syntax is enabled (using the {**$X+**} compiler directive), a value of type PChar can be indexed with a single index expression of type Word. The index expression specifies an *offset* to add to the character pointer before it is dereferenced to produce a *Char* type variable reference.

Records and field designators

A specific field of a record variable is denoted by a variable reference that refers to the record variable, followed by a field designator specifying the field.



Some examples of a field designator include the following:

```
Today.Year
Results[1].Count
Results[1].When.Month
```

In a statement within a **with** statement, a field designator doesn't have to be preceded by a variable reference to its containing record.

Object component designators

The format of an object component designator is the same as that of a record field designator; that is, it consists of an instance (a variable reference), followed by a period and a component identifier. A component designator that designates a method is called a

method designator. A **with** statement can be applied to an instance of an object type. In that case, the instance and the period can be omitted in referencing components of the object type.

The instance and the period can also be omitted within any method block, and when they are, the effect is the same as if *Self* and a period was written before the component reference.

Pointers and dynamic variables

The value of a pointer variable is either **nil** or the address of a value that points to a dynamic variable.

The dynamic variable pointed to by a pointer variable is referenced by writing the pointer symbol (^) after the pointer variable.

You create dynamic variables and their pointer values with the standard procedures *New* and *GetMem*. You can use the @ (address-of) operator and the standard function *Ptr* to create pointer values that are treated as pointers to dynamic variables.

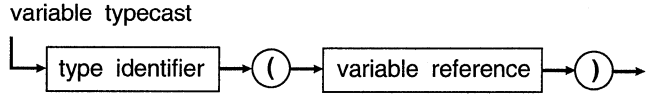
nil does not point to any variable. The results are undefined if you access a dynamic variable when the pointer's value is **nil** or undefined.

Some examples of references to dynamic variables:

```
P1^
P1^.Sibling^
Results[1].Data^
```

Variable typecasts

A variable reference of one type can be changed into a variable reference of another type through a *variable typecast*.



When a variable typecast is applied to a variable reference, the variable reference is treated as an instance of the type specified by the type identifier. The size of the variable (the number of bytes

occupied by the variable) must be the same as the size of the type denoted by the type identifier. A variable typecast can be followed by one or more qualifiers, as allowed by the specified type.

Some examples of variable typecasts follow:

```
type
  ByteRec = record
    Lo, Hi: Byte;
  end;
  WordRec = record
    Low, High: Word;
  end;
  PtrRec  = record
    Ofs, Seg: Word;
  end;
  BytePtr = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;
begin
  W := $1234;
  B := ByteRec(W).Lo;
  ByteRec(W).Hi := 0;
  L := $01234567;
  W := WordRec(L).Low;
  B := ByteRec(WordRec(L).Low).Hi;
  B := BytePtr(L)^;
  P := Ptr($40,$49);
  W := PtrRec(P).Seg;
  Inc(PtrRec(P).Ofs, 4);
end.
```

Notice the use of the *ByteRec* type to access the low- and high-order bytes of a word; this corresponds to the built-in functions *Lo* and *Hi*, except that a variable typecast can also be used on the left hand side of an assignment. Also, observe the use of the *WordRec* and *PtrRec* types to access the low- and high-order words of a long integer, and the offset and segment parts of a pointer.

Turbo Pascal fully supports variable typecasts involving procedural types. For example, given the declarations

```
type
  Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can construct the following assignments:

```
F := Func(P);                    { Assign procedural value in P to F }
Func(P) := F;                    { Assign procedural value in F to P }
@F := P;                           { Assign pointer value in P to F }
P := @F;                           { Assign pointer value in F to P }
N := F(N);                                   { Call function via F }
N := Func(P)(N);                             { Call function via P }
```

In particular, notice that the address operator (**@**), when applied to a procedural variable, can be used on the left-hand side of an assignment. Also, notice the typecast on the last line to call a function via a pointer variable.

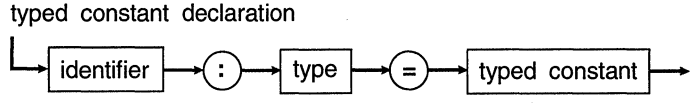
C H A P T E R

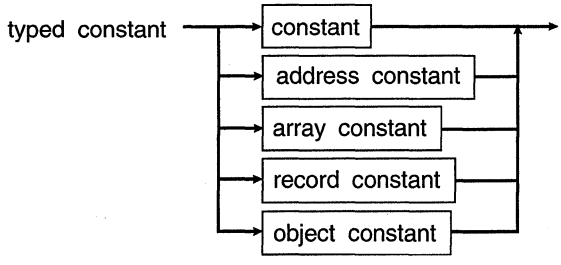
5

Typed constants

See the section entitled "Constant declarations" in Chapter 1.

Typed constants can be compared to initialized variables—variables whose values are defined on entry to their block. Unlike an untyped constant, the declaration of a typed constant specifies both the type and the value of the constant.





Typed constants can be used exactly like variables of the same type, and can appear on the left-hand side in an assignment statement. Note that typed constants are initialized *only once*—at the beginning of a program. Thus, for each entry to a procedure or function, the locally declared typed constants are not reinitialized.

In addition to a normal constant expression, the value of a typed constant may be specified using a *constant address expression*. A constant address expression is an expression that involves taking the address, offset, or segment of a global variable, a typed constant, a procedure, or a function. Constant address expressions cannot reference local variables or dynamic (heap based) variables, since their addresses cannot be computed at compile-time.

Simple-type constants

Declaring a typed constant as a simple type simply specifies the value of the constant:

```
const
  Maximum: Integer = 9999;
  Factor: Real = -0.1;
  Breakchar: Char = #3;
```

As mentioned earlier, the value of a typed constant may be specified using a constant address expression, that is, an expression that takes the address, offset, or segment of a global variable, a typed constant, a procedure, or a function. For example,

```
var
  Buffer: array[0..1023] of Byte;
const
  BufferOfs: Word = Ofs(Buffer);
  BufferSeg: Word = Seg(Buffer);
```

Because a typed constant is actually a variable with a constant value, it cannot be interchanged with ordinary constants. For instance, it cannot be used in the declaration of other constants or types.

```
const
  Min: Integer = 0;
  Max: Integer = 99;
type
  Vector = array[Min..Max] of Integer;
```

The *Vector* declaration is invalid, because *Min* and *Max* are typed constants.

String-type constants

The declaration of a typed constant of a string type specifies the maximum length of the string and its initial value:

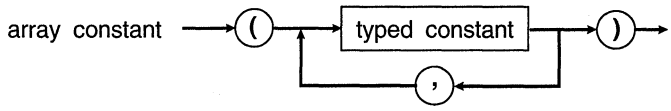
```
const
  Heading: string[7] = 'Section';
  NewLine: string[2] = #13#10;
  TrueStr: string[5] = 'Yes';
  FalseStr: string[5] = 'No';
```

Structured-type constants

The declaration of a structured-type constant specifies the value of each of the structure's components. Turbo Pascal supports the declaration of type array, record, set, and pointer constants; type file constants, and constants of array and record types that contain type file components are not allowed.

Array-type constants

The declaration of an array-type constant specifies, enclosed in parentheses and separated by commas, the values of the components.



An example of an array-type constant follows:

```
type
  Status = (Active, Passive, Waiting);
  StatusMap = array[Status] of string[7];
const
  StatStr: StatusMap = ('Active', 'Passive', 'Waiting');
```

This example defines the array constant *StatStr*, which can be used to convert values of type *Status* into their corresponding string representations. The components of *StatStr* are

```
StatStr[Active] = 'Active'
StatStr[Passive] = 'Passive'
StatStr[Waiting] = 'Waiting'
```

The component type of an array constant can be any type except a file type. Packed string-type constants (character arrays) can be specified both as single characters and as strings. The definition

```
const
  Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5',
    '6', '7', '8', '9');
```

can be expressed more conveniently as

```
const
  Digits: array[0..9] of Char = '0123456789';
```

When the extended syntax is enabled (using a {**$X+**} compiler directive), a zero-based character array can be initialized with a string that is shorter than the declared length of the array. For example,

```
const
  FileName = array[0..79] of Char = 'TEST.PAS';
```

For details about null-terminated strings, look in Chapter 13.

In such cases, the remaining characters are set to NULL (#0) and the array will effectively contain a null-terminated string.

Multidimensional array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas. The innermost constants correspond to the rightmost dimensions. The declaration

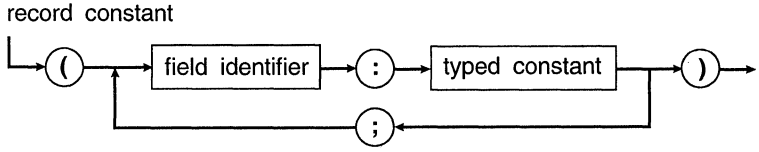
```
type
  Cube = array[0..1, 0..1, 0..1] of Integer;
const
  Maze: Cube = (((0, 1), (2, 3)), ((4, 5), (6, 7)));
```

provides an initialized array *Maze* with the following values:

```
Maze[0, 0, 0] = 0
Maze[0, 0, 1] = 1
Maze[0, 1, 0] = 2
Maze[0, 1, 1] = 3
Maze[1, 0, 0] = 4
Maze[1, 0, 1] = 5
Maze[1, 1, 0] = 6
Maze[1, 1, 1] = 7
```

Record-type constants

The declaration of a record-type constant specifies the identifier and value of each field, enclosed in parentheses and separated by semicolons.



Some examples of record constants follow:

```
type
  Point = record
    X, Y: Real;
  end;
  Vector = array[0..1] of Point;
  Month = (Jan, Feb, Mar, Apr, May, Jun, Jly, Aug, Sep, Oct,
    Nov, Dec);
  Date = record
    D: 1..31;
    M: Month;
    Y: 1900..1999;
  end;
const
  Origin: Point = (X: 0.0; Y: 0.0);
  Line: Vector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: Date = (D: 2; M: Dec; Y: 1960);
```

The fields must be specified in the same order as they appear in the definition of the record type. If a record contains fields of file types, the constants of that record type cannot be declared. If a record contains a variant, only fields of the selected variant can be specified. If the variant contains a tag field, then its value must be specified.

Object-type constants

The declaration of an object-type constant uses the same syntax as the declaration of a record-type constant. No value is, or can be, specified for method components. Referring to the earlier object-

type declarations, here are some examples of object-type constants:

```
const
  ZeroPoint: Point = (X: 0; Y: 0);
  ScreenRect: Rect = (A: (X: 0; Y: 0); B: (X: 80; Y: 25));
  CountField: NumField = (X: 5; Y: 20; Len: 4; Name: nil;
    Value: 0; Min: -999; Max: 999);
```

Constants of an object type that contains virtual methods need *not* be initialized through a constructor call—this initialization is handled automatically by the compiler.

Set-type constants

Just like a simple-type constant, the declaration of a set-type constant specifies the value of the set using a constant expression. Some examples follow:

```
type
  Digits = set of 0..9;
  Letters = set of 'A'..'Z';
const
  EvenDigits: Digits = [0, 2, 4, 6, 8];
  Vowels: Letters = ['A', 'E', 'I', 'O', 'U', 'Y'];
  HexDigits: set of '0'..'z' = ['0'..'9', 'A'..'F', 'a'...f'];
```

Pointer-type constants

The declaration of a pointer-type constant typically uses a constant address expression to specify the pointer value. Some examples follow:

```
type
  Direction = (Left, Right, Up, Down);
  StringPtr = ^String;
  NodePtr = ^Node;
  Node = record
    Next: NodePtr;
    Symbol: StringPtr;
    Value: Direction;
  end;
const
  S1: string[4] = 'DOWN';
  S2: string[2] = 'UP';
  S3: string[5] = 'RIGHT';
```

```
S4: string[4] = 'LEFT';
N1: Node = (Next: nil; Symbol: @S1; Value: Down);
N2: Node = (Next: @N1; Symbol: @S2; Value: Up);
N3: Node = (Next: @N2; Symbol: @S3; Value: Right);
N4: Node = (Next: @N3; Symbol: @S4; Value: Left);
DirectionTable: NodePtr = @N4;
```

When the extended syntax is enabled (using a {**$X+**} compiler directive), a typed constant of type PChar can be initialized with a string constant. For example,

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

The result is that the pointer now points to an area of memory that contains a zero-terminated copy of the string literal. See Chapter 13, "The Strings unit" for more information.

Procedural-type constants

A procedural-type constant must specify the identifier of a procedure or function that is assignment compatible with the type of the constant. An example follows:

```
type
  ErrorProc = procedure(ErrorCode: Integer);

procedure DefaultError(ErrorCode: Integer); far;
begin
  WriteLn('Error ', ErrorCode, '.');
end;

const
  ErrorHandler: ErrorProc = DefaultError;
```

C H A P T E R

6

Expressions

Expressions are made up of *operators* and *operands*. Most Pascal operators are *binary*, that is, they take two operands; the rest are *unary* and take only one operand. Binary operators use the usual algebraic form, for example, $A + B$. A unary operator always precedes its operand, for example, $-B$.

In more complex expressions, rules of precedence clarify the order in which operations are performed (see the following table).

Table 6.1
Precedence of operators

Operators	Precedence	Categories
@, not	first (high)	unary operators
***, /, div, mod, and, shl, shr**	second	multiplying operators
+,-, or, xor	third	adding operators
=, <>, <, >, <=, >=, in	fourth (low)	relational operators

There are three basic rules of precedence:

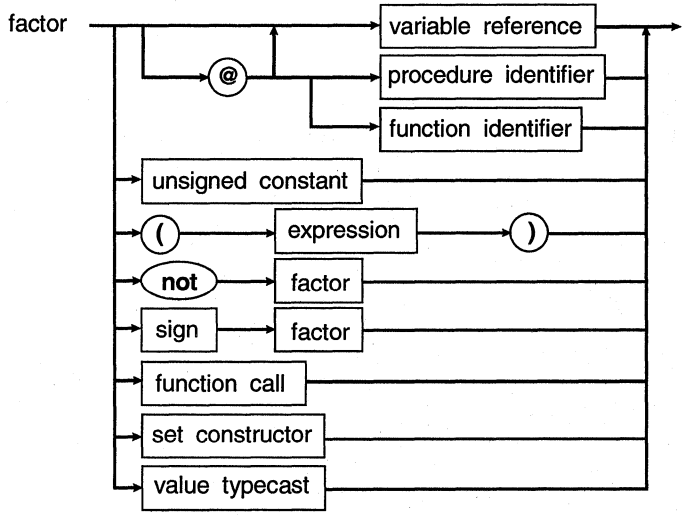
1. An operand between two operators of different precedence is bound to the operator with higher precedence.
2. An operand between two equal operators is bound to the one on its left.
3. Expressions within parentheses are evaluated prior to being treated as a single operand.

Operations with equal precedence are normally performed from left to right, although the compiler may at times rearrange the operands to generate optimum code.

Expression syntax

The precedence rules follow from the syntax of expressions, which are built from factors, terms, and simple expressions.

A factor's syntax follows:



See "Function calls" on page 77.

A function call activates a function and denotes the value returned by the function.

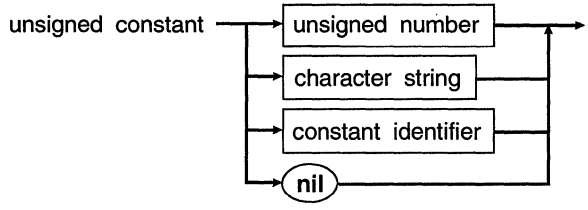
See "Set constructors" on page 78.

A set constructor denotes a value of a set type.

See "Value typecasts" on page 79.

A value typecast changes the type of a value.

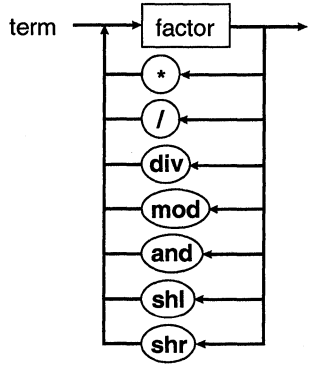
An unsigned constant has the following syntax:



Some examples of factors include the following:

```
X                                    { Variable reference }
@X                                 { Pointer to a variable }
15                                    { Unsigned constant }
(X + Y + Z)                              { Subexpression }
Sin(X / 2)                               { Function call }
exit['0'..'9', 'A'..'Z']               { Set constructor }
not Done                          { Negation of a Boolean }
Char(Digit + 48)                        { Value typecast }
```

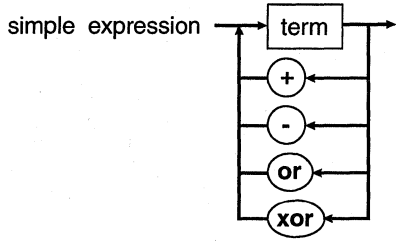
Terms apply the multiplying operators to factors:



Here are some examples of terms:

```
X * Y
Z / (1 - Z)
Done or Error
(X <= Y) and (Y < Z)
```

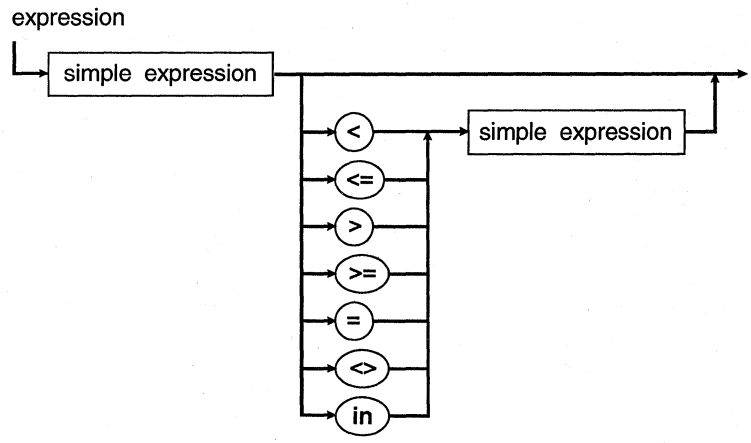
Simple expressions apply adding operators and signs to terms:



Here are some examples of simple expressions:

```
X + Y
-X
Hue1 + Hue2
I * J + 1
```

An expression applies the relational operators to simple expressions:



Here are some examples of expressions:

```
X = 1.5
Done <> Error
(I < J) = (J < K)
C in Hue1
```

Operators

The operators are classified as arithmetic operators, logical operators, string operators, set operators, relational operators, and the @ operator.

Arithmetic operators

The following tables show the types of operands and results for binary and unary arithmetic operations.

Table 6.2
Binary arithmetic operations

*The + operator is also used as a string or set operator, and the +, –, and * operators are also used as set operators.*

Operator	Operation	Operand types	Result type
+	addition	integer type real type	integer type real type
–	subtraction	integer type real type	integer type real type
*	multiplication	integer type real type	integer type real type
/	division	integer type real type	real type real type
div	integer division	integer type	integer type
mod	remainder	integer type	integer type

Table 6.3
Unary arithmetic operations

Operator	Operation	Operand types	Result type
+	sign identity	integer type real type	integer type real type
–	sign negation	integer type real type	integer type real type

Any operand whose type is a subrange of an ordinal type is treated as if it were of the ordinal type.

See the section "Integer types" in Chapter 3 for a definition of common types.

If both operands of a **+**, **–**, *****, **div**, or **mod** operator are of an integer type, the result type is of the common type of the two operands.

If one or both operands of a **+**, **–**, or ***** operator are of a real type, the type of the result is Real in the {**$N-**} state or Extended in the {**$N+**} state.

If the operand of the sign identity or sign negation operator is of an integer type, the result is of the same integer type. If the operator is of a real type, the type of the result is Real or Extended.

The value of *X* / *Y* is always of type Real or Extended regardless of the operand types. An error occurs if *Y* is zero.

The value of *I* **div** *J* is the mathematical quotient of *I* / *J*, rounded in the direction of zero to an integer-type value. An error occurs if *J* is zero.

The **mod** operator returns the remainder obtained by dividing its two operands, that is,

```
I mod J = I - (I div J) * J
```

The sign of the result of **mod** is the same as the sign of *I*. An error occurs if *J* is zero.

Logical operators

The types of operands and results for logical operations are shown in Table 6.4.

Table 6.4
Logical operations
*The **not** operator is a unary operator.*

Operator	Operation	Operand types	Result type
not	bitwise negation	integer type	integer type
and	bitwise and	integer type	integer type
or	bitwise or	integer type	integer type
xor	bitwise xor	integer type	integer type
shl	shift left	integer type	integer type
shr	shift right	integer type	integer type

If the operand of the **not** operator is of an integer type, the result is of the same integer type.

If both operands of an **and**, **or**, or **xor** operator are of an integer type, the result type is the common type of the two operands.

The operations *I* **shl** *J* and *I* **shr** *J* shift the value of *I* to the left or to the right by *J* bits. The type of the result is the same as the type of *I*.

Boolean operators

The types of operands and results for Boolean operations are shown in Table 6.5.

Table 6.5
Boolean operations

*The **not** operator is a unary operator.*

Operator	Operation	Operand types	Result type
not	negation	Boolean	Boolean
and	logical and	Boolean	Boolean
or	logical or	Boolean	Boolean
xor	logical xor	Boolean	Boolean

Normal Boolean logic governs the results of these operations. For instance, *A* **and** *B* is True only if both *A* and *B* are True.

Turbo Pascal supports two different models of code generation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation.

Complete evaluation means that every operand of a Boolean expression, built from the **and** and **or** operators, is guaranteed to be evaluated, even when the result of the entire expression is already known. This model is convenient when one or more operands of an expression are functions with side effects that alter the meaning of the program.

Short-circuit evaluation guarantees strict left-to-right evaluation and that evaluation stops as soon as the result of the entire expression becomes evident. This model is convenient in most cases, since it guarantees minimum execution time, and usually minimum code size. Short-circuit evaluation also makes possible the evaluation of constructs that would not otherwise be legal; for instance:

```
while (I <= Length(S)) and (S[I] <> ' ') do
  Inc(I);
while (P <> nil) and (P^.Value <> 5) do
  P := P^.Next;
```

In both cases, the second test is not evaluated if the first test is False.

The evaluation model is controlled through the **$B** compiler directive. The default state is {**$B-**} (unless changed using the **O**ptions | **C**ompiler menu), and in this state short-circuit evaluation code is generated. In the {**$B+**} state, complete evaluation code is generated.

Since standard Pascal does not specify which model should be used for Boolean expression evaluation, programs depending on either model being in effect are not truly portable. However, sacrificing portability is often worth gaining the execution speed and simplicity provided by the short-circuit model.

String operator

The types of operands and results for string operation are shown in Table 6.6.

Table 6.6
String operation

Operator	Operation	Operand types	Result type
+	concatenation	string type, Char type, or packed string type	string type

Turbo Pascal allows the **+** operator to be used to concatenate two string operands. The result of the operation *S* **+** *T*, where *S* and *T* are of a string type, a Char type, or a packed string type, is the concatenation of *S* and *T*. The result is compatible with any string type (but not with Char types and packed string types). If the resulting string is longer than 255 characters, it is truncated after character 255.

PChar operators

The extended syntax (enabled using a {**$X+**} compiler directive) supports a number of new operations on character pointers. The plus (+) and minus (-) operators can be used to increment and decrement the offset part of a pointer value, and the minus operator can be used to calculate the distance (difference) between the offset parts of two character pointers. Assuming that *P* and *Q* are values of type PChar and *I* is a value of type Word, these constructs are allowed:

P + I	Add I to the offset part of P
I + P	Add I to the offset part of P
P – I	Subtract I from the offset part of P
P – Q	Subtract offset part of Q from offset part of P

The operations *P* + *I* and *I* + *P* adds *I* to the address given by *P*, producing a pointer that points *I* characters after *P*. The operation *P* – *I* subtracts *I* from the address given by *P*, producing a pointer that points *I* characters before *P*.

The operation *P* – *Q* computes the distance between *Q* (the lower address) and *P* (the higher address), resulting in a value of type Word that gives the number of characters between *Q* and *P*. This operation assumes that *P* and *Q* point within the same character

array. If the two character pointers point into different character arrays, the result is undefined.

Set operators

The types of operands for set operations are shown in Table 6.7.

Table 6.7
Set operations

Operator	Operation	Operand types
+	union	compatible set types
–	difference	compatible set types
*	intersection	compatible set types

The results of set operations conform to the rules of set logic:

- An ordinal value *C* is in *A* + *B* only if *C* is in *A* or *B*.
- An ordinal value *C* is in *A* – *B* only if *C* is in *A* and not in *B*.
- An ordinal value *C* is in *A* * *B* only if *C* is in both *A* and *B*.

If the smallest ordinal value that is a member of the result of a set operation is *A* and the largest is *B*, then the type of the result is **set of** *A..B*.

Relational operators

The types of operands and results for relational operations are shown in Table 6.8.

Table 6.8
Relational operations

Operator type	Operation	Operand types	Result type
=	equal	compatible simple, pointer, set, string, or packed string types	Boolean
<>	not equal	compatible simple, pointer, set, string, or packed string types	Boolean
<	less than	compatible simple, string, packed string types, or PChar	Boolean
>	greater than	compatible simple, string, packed string types, or PChar	Boolean
<=	less or equal	compatible simple, string, packed string types, or PChar	Boolean

Table 6.8: Relational operations (continued)

>=	greater or equal	compatible simple, string, or packed string types, or PChar	Boolean
<=	subset of	compatible set types	Boolean
>=	superset of	compatible set types	Boolean
in	member of	left operand: any ordinal type *T*; right operand: set whose base is compatible with *T*.	Boolean

Comparing simple types

When the operands of **=**, **<>**, **<**, **>**, **>=**, or **<=** are of simple types, they must be compatible types; however, if one operand is of a real type, the other can be of an integer type.

Comparing strings

The relational operators **=**, **<>**, **<**, **>**, **>=**, and **<=** compare strings according to the ordering of the extended ASCII character set. Any two string values can be compared, because all string values are compatible.

A character-type value is compatible with a string-type value, and when the two are compared, the character-type value is treated as a string-type value with length 1. When a packed string-type value with *N* components is compared with a string-type value, it is treated as a string-type value with length *N*.

Comparing packed strings

The relational operators **=**, **<>**, **<**, **>**, **>=**, and **<=** can also be used to compare two packed string-type values if both have the same number of components. If the number of components is *N*, then the operation corresponds to comparing two strings, each of length *N*.

Comparing pointers

The operators **=** and **<>** can be used on compatible pointer-type operands. Two pointers are equal only if they point to the same object.

When it compares pointers, Turbo Pascal simply compares the segment and offset parts. Because of the segment mapping scheme of the 80x86 processors, two logically different pointers can in fact point to the same physical memory location. For instance, *Ptr*($0040,$0049) and *Ptr*($0000,$0449) are two pointers

to the same physical address. Pointers returned by the standard procedures *New* and *GetMem* are always normalized (offset part in the range $0000 to $000F), and will therefore always compare correctly. When creating pointers with the *Ptr* standard function, special care must be taken if such pointers are to be compared.

Comparing character pointers

The extended syntax (enabled using a {**$X+**} compiler directive) allows the >, <, >=, and <= operators to be applied to PChar values. Note, however, that these relational tests assume that the two pointers being compared *point within the same character array*, and for that reason, the operators only compare the offset parts of the two pointer values. If the two character pointers point into different character arrays, the result is undefined.

Comparing sets

If *A* and *B* are set operands, their comparisons produce these results:

- *A* **=** *B* is True only if *A* and *B* contain exactly the same members; otherwise, *A* **<>** *B*.
- *A* **<=** *B* is True only if every member of *A* is also a member of *B*.
- *A* **>=** *B* is True only if every member of *B* is also a member of *A*.

Testing set membership

The **in** operator returns True when the value of the ordinal-type operand is a member of the set-type operand; otherwise, it returns False.

The @ operator

A pointer to a variable can be created with the **@** operator. Table 6.9 shows the operand and result types.

Table 6.9
Pointer operation

Operator	Operation	Operand types	Result type
@	Pointer formation	Variable reference or procedure or function identifier	*Pointer* (same as **nil**)

Special rules apply to use of the @ operator with a procedural variable. For further details, see "Procedural types" on page 111.

@ is a unary operator that takes a variable reference or a procedure or function identifier as its operand, and returns a pointer to the operand. The type of the value is the same as the type of **nil**, therefore it can be assigned to any pointer variable.

@ with a variable

The use of @ with an ordinary variable (not a parameter) is uncomplicated. Given the declarations

```
type
  TwoChar = array[0..1] of Char;
var
  Int: Integer;
  TwoCharPtr: ^TwoChar;
```

then the statement

```
TwoCharPtr := @Int;
```

causes *TwoCharPtr* to point to *Int*. *TwoCharPtr^* becomes a re-interpretation of the value of *Int*, as though it were an `array[0..1] of Char`.

@ with a value parameter

Applying @ to a formal value parameter results in a pointer to the stack location containing the actual value. Suppose *Foo* is a formal value parameter in a procedure and *FooPtr* is a pointer variable. If the procedure executes the statement

```
FooPtr := @Foo;
```

then *FooPtr^* references *Foo*'s value. However, *FooPtr^* does not reference *Foo* itself, rather it references the value that was taken from *Foo* and stored on the stack.

@ with a variable parameter

Applying @ to a formal variable parameter results in a pointer to the actual parameter (the pointer is taken from the stack). Suppose *One* is a formal variable parameter of a procedure, *Two* is a variable passed to the procedure as *One*'s actual parameter, and *OnePtr* is a pointer variable. If the procedure executes the statement

```
OnePtr := @One;
```

then *OnePtr* is a pointer to *Two*, and *OnePtr^* is a reference to *Two* itself.

@ with a procedure or function

You can apply @ to a procedure or a function to produce a pointer to its entry point. Turbo Pascal does not give you a mechanism for using such a pointer. The only use for a procedure pointer is to pass it to an assembly language routine or to use it in an **inline** statement. See "Turbo Assembler and Turbo Pascal" on

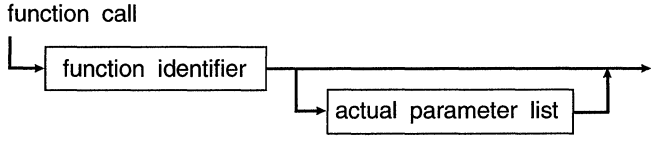
page 292 for information on interfacing Turbo Assembler and Turbo Pascal.

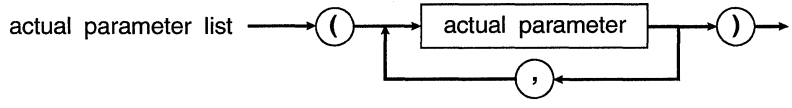
@ with a method You can apply @ to a qualified method identifier to produce a pointer to the method's entry point.

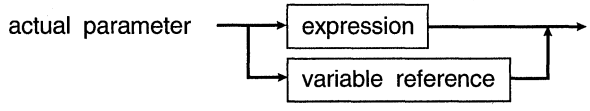
Function calls

A function call activates the function specified by the function identifier. Any identifier declared to denote a function is a function identifier.

The function call must have a list of actual parameters if the corresponding function declaration contains a list of formal parameters. Each parameter takes the place of the corresponding formal parameter according to parameter rules set forth in Chapter 19, "Input and output issues."







Some examples of function calls follow:

A function can also be invoked via a procedural variable. For further details, refer to the "Procedural types" section on page 111.

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
```

The syntax of a function call has been extended to allow a method designator or a qualified method identifier denoting a function to replace the function identifier.

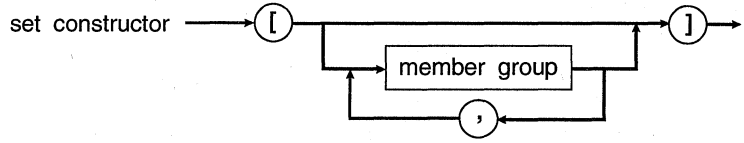
The discussion of extensions to procedure statements in the section, "Procedure statements" in Chapter 7 also applies to function calls.

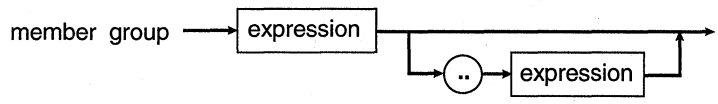
See "Extended syntax" in Chapter 21 for details.

In the extended syntax (**$X+**) mode, function calls can be used as statements; that is, the result of a function call can be discarded.

Set constructors

A set constructor denotes a set-type value, and is formed by writing expressions within brackets ([]). Each expression denotes a value of the set.





The notation [ ] denotes the empty set, which is assignment-compatible with every set type. Any member group *X..Y* denotes as set members all values in the range *X..Y*. If *X* is greater than *Y*, then *X..Y* does not denote any members and [*X..Y*] denotes the empty set.

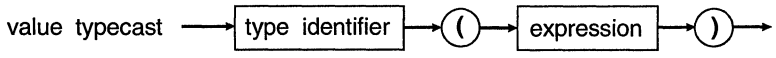
All expression values in member groups in a particular set constructor must be of the same ordinal type.

Some examples of set constructors follow:

```
[red, C, green]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

Value typecasts

The type of an expression can be changed to another type through a value typecast.



The expression type and the specified type must both be either ordinal types or pointer types. For ordinal types, the resulting value is obtained by converting the expression. The conversion may involve truncation or extension of the original value if the size of the specified type is different from that of the expression. In cases where the value is extended, the sign of the value is always preserved; that is, the value is sign-extended.

See "Variable typecasts" on page 53.

The syntax of a value typecast is almost identical to that of a variable typecast. However, value typecasts operate on values, not on variables, and can therefore not participate in variable references; that is, a value typecast may not be followed by qualifiers. In particular, value typecasts cannot appear on the left-hand side of an assignment statement.

Some examples of value typecasts include the following:

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
BytePtr(Ptr($40, $49))
```

Procedural types in expressions

In general, the use of a procedural variable in a statement or an expression denotes a call of the procedure or function stored in

the variable. There is however one exception: When Turbo Pascal sees a procedural variable on the left-hand side of an assignment statement, it knows that the right-hand side has to represent a procedural value. For example, consider the following program:

```
type
  IntFunc = function: Integer;

var
  F: IntFunc;
  N: Integer;

function ReadInt: Integer; far;
var
  I: Integer;
begin
  Read(I);
  ReadInt := I;
end;

begin
  F := ReadInt;                          { Assign procedural value }
  N := ReadInt;                           { Assign function result }
end.
```

The first statement in the main program assigns the procedural value (address of) *ReadInt* to the procedural variable *F*, where the second statement calls *ReadInt*, and assigns the returned value to *N*. The distinction between getting the procedural value or calling the function is made by the type of the variable being assigned (*F* or *N*).

Unfortunately, there are situations where the compiler cannot determine the desired action from the context. For example, in the following statement, there is no obvious way the compiler can know if it should compare the procedural value in *F* to the procedural value of *ReadInt*, to determine if *F* currently points to *ReadInt*, or whether it should call *F* and *ReadInt*, and then compare the returned values.

```
if F = ReadInt then
  WriteLn('Equal');
```

However, standard Pascal syntax specifies that the occurrence of a function identifier in an expression denotes a call to that function, so the effect of the preceding statement is to call *F* and *ReadInt*, and then compare the returned values. To compare the procedural value in *F* to the procedural value of *ReadInt*, the following construct must be used:

```
if @F = @ReadInt then
  Writeln('Equal');
```

When applied to a procedural variable or a procedure or function identifier, the address (**@**) operator prevents the compiler from calling the procedure, and at the same time converts the argument into a pointer. Thus, *@F* converts *F* into an untyped pointer variable that contains an address, and *@ReadInt* returns the address of *ReadInt*; the two pointer values can then be compared to determine if *F* currently refers to *ReadInt*.

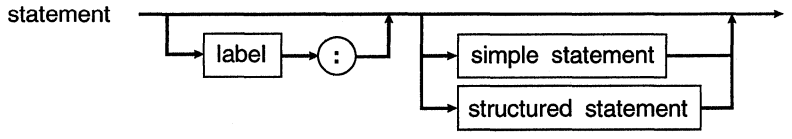
To get the memory address of a procedural variable, rather than the address stored in it, a double address (**@@**) operator must be used. For example, where *@P* means convert *P* into an untyped pointer variable, *@@P* means return the physical address of the variable *P*.

C H A P T E R

7

Statements

Statements describe algorithmic actions that can be executed. Labels can prefix statements, and these labels can be referenced by **goto** statements.

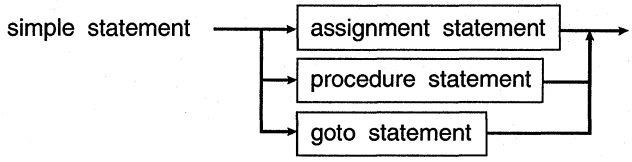


As you saw in Chapter 1, a label is either a digit sequence in the range 0 to 9999 or an identifier.

There are two main types of statements: simple statements and structured statements.

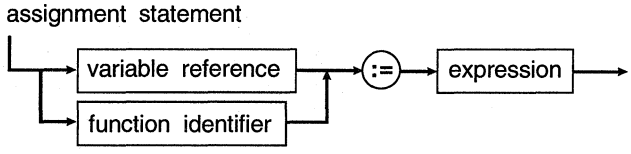
Simple statements

A *simple* statement is a statement that doesn't contain any other statements.



Assignment statements

Assignment statements either replace the current value of a variable with a new value specified by an expression or specify an expression whose value is to be returned by a function.



See the section "Type compatibility" on page 44.

The expression must be assignment-compatible with the type of the variable or the result type of the function.

Some examples of assignment statements follow:

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I * K;
```

Object type assignments

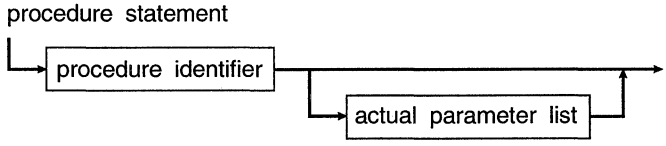
Object types are discussed in more detail in Chapter 5.

The rules of object type assignment compatibility allow an instance of an object type to be assigned an instance of any of its descendant types. Such an assignment constitutes a *projection* of the descendant onto the space spanned by its ancestor. For example, given an instance *F* of type *Field*, and an instance Z of type *ZipField*, the assignment *F* := Z will copy only the fields *X*, *Y*, *Len*, and *Name*.

Assignment to an instance of an object type does *not* entail initialization of the instance. Referring to the preceding example, the assignment *F* := Z does not mean that a constructor call for *F* can be omitted.

Procedure statements

A **procedure** statement specifies the activation of the procedure denoted by the procedure identifier. If the corresponding procedure declaration contains a list of formal parameters, then the procedure statement must have a matching list of actual parameters (parameters listed in definitions are *formal* parameters; in the calling statement, they are *actual* parameters). The actual parameters are passed to the formal parameters as part of the call.



A procedure can also be invoked via a procedural variable. For further details, refer to the "Procedural types" section on page 111.

Some examples of procedure statements follow:

```
PrintHeading;
Transpose(A, N, M);
Find(Name, Address);
```

Method, constructor, and destructor calls

The syntax of a procedure statement has been extended to allow a method designator denoting a procedure, constructor, or destructor to replace the procedure identifier.

The instance denoted by the method designator serves two purposes. First, in the case of a virtual method, the *actual* (run time) type of the instance determines which implementation of the method is activated. Second, the instance itself becomes an implicit actual parameter of the method; it corresponds to a formal variable parameter named *Self* that possesses the type corresponding to the activated method.

Within a method, a procedure statement allows a qualified method identifier to denote activation of a specific method. The object type given in the qualified identifier must be the same as the method's object type, or an ancestor of it. This type of activation is called a *qualified activation*.

The implicit *Self* parameter of a qualified activation becomes the *Self* of the method containing the call. A qualified activation never

employs the virtual method dispatch mechanism—the call is always static and always invokes the specified method.

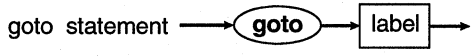
A qualified activation is generally used within an override method to activate the overridden method. Referring to the types declared earlier, here are some examples of qualified activations:

```
constructor NumField.Init(FX, FY, FLen: Integer;
  FName: String; FMin, FMax: Longint);
begin
  Field.Init(FX, FY, FLen, FName);
  Value := 0;
  Min := FMin;
  Max := FMax;
end;

function ZipField.PutStr(S: String): Boolean;
begin
  PutStr := (Length(S) = 5) and NumField.PutStr(S);
end;
```

As these examples demonstrate, a qualified activation allows an override method to "reuse" the code of the method it overrides.

Goto statements

A **goto** statement transfers program execution to the statement prefixed by the label referenced in the **goto** statement. The syntax diagram of a **goto** statement follows:

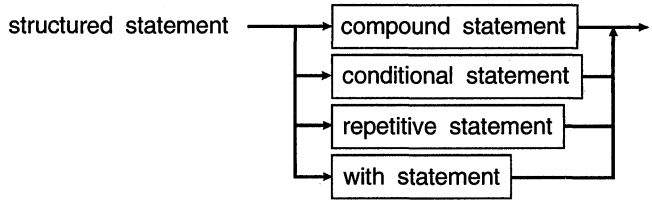


The following rules should be observed when using **goto** statements:

- The label referenced by a **goto** statement must be in the same block as the **goto** statement. In other words, it is not possible to jump into or out of a procedure or function.
- Jumping into a structured statement from outside that structured statement (that is, jumping to a "deeper" level of nesting) can have undefined effects, although the compiler will not indicate an error.

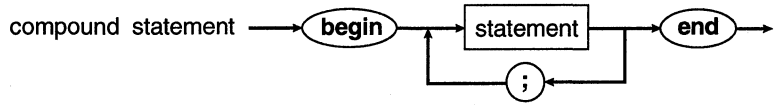
Structured statements

Structured statements are constructs composed of other statements that are to be executed in sequence (compound and **with** statements), conditionally (conditional statements), or repeatedly (repetitive statements).



Compound statements

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The component statements are treated as one statement, crucial in contexts where the Pascal syntax only allows one statement. **begin** and **end** bracket the statements, which are separated by semicolons.

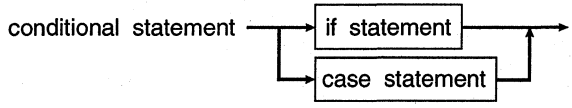


Here's an example of a compound statement:

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

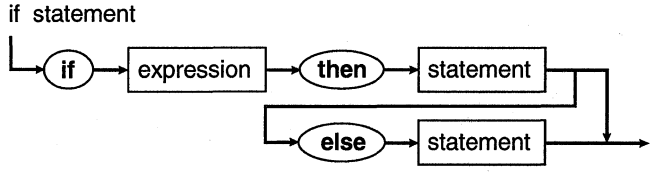
Conditional statements

A conditional statement selects for execution a single one (or none) of its component statements.



If statements

The syntax for an **if** statement reads like this:



The expression must yield a result of the standard type Boolean. If the expression produces the value True, then the statement following **then** is executed.

If the expression produces False and the **else** part is present, the statement following **else** is executed; if the **else** part is not present, nothing is executed.

The syntactic ambiguity arising from the construct

```
if e1 then if e2 then s1 else s2;
```

is resolved by interpreting the construct as follows:

```
if e1 then
begin
  if e2 then
    s1
  else
    s2
end;
```

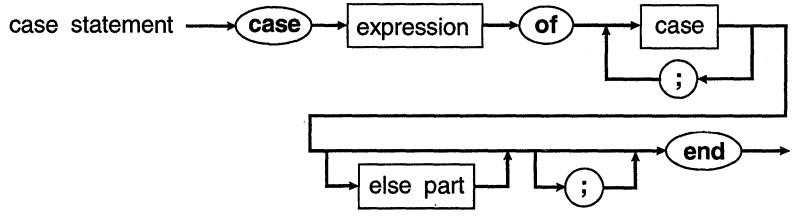
In general, an **else** is associated with the closest **if** not already associated with an **else**.

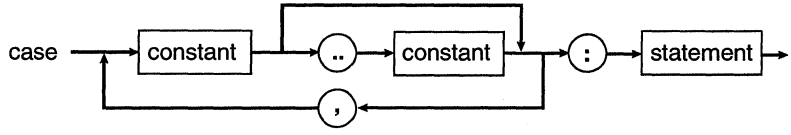
Two examples of **if** statements follow:

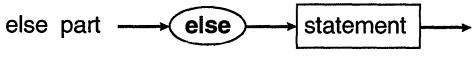
```
if X < 1.5 then
  Z := X + Y
else
  Z := 1.5;

if P1 <> nil then
  P1 := P1^.Father;
```

Case statements

The **case** statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called case constants) or with the word **else**. The selector must be of a byte-sized or word-sized ordinal type. Thus, string types and the integer type Longint are invalid selector types. All **case** constants must be unique and of an ordinal type compatible with the selector type.







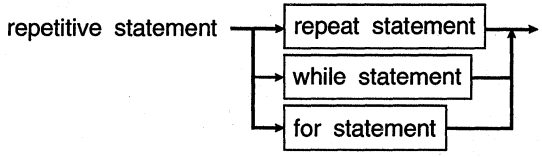
The **case** statement executes the statement prefixed by a **case** constant equal to the value of the selector or a **case** range containing the value of the selector. If no such **case** constant of the **case** range exists and an **else** part is present, the statement following **else** is executed. If there is no **else** part, nothing is executed.

Examples of **case** statements include

```
case Operator of
  Plus: X := X + Y;
  Minus: X := X - Y;
  Times: X := X * Y;
end;

case I of
  0, 2, 4, 6, 8: Writeln('Even digit');
  1, 3, 5, 7, 9: Writeln('Odd digit');
  10..100: Writeln('Between 10 and 100');
else
  Writeln('Negative or greater than 100');
end;
```

Repetitive statements

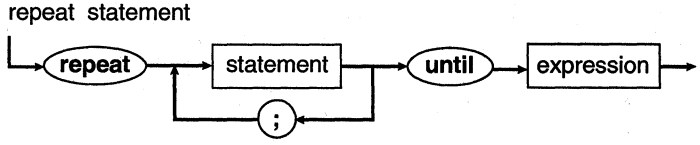
Repetitive statements specify certain statements to be executed repeatedly.



If the number of repetitions is known beforehand, the **for** statement is the appropriate construct. Otherwise, the **while** or **repeat** statement should be used.

Repeat statements

A **repeat** statement contains an expression that controls the repeated execution of a statement sequence within that **repeat** statement.



The expression must produce a result of type Boolean. The statements between the symbols **repeat** and **until** are executed in

sequence until, at the end of a sequence, the expression yields True. The sequence is executed at least once, because the expression is evaluated *after* the execution of each sequence.

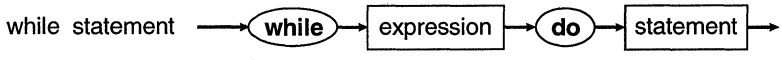
Examples of **repeat** statements follow:

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;
```

```
repeat
  Write('Enter value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While statements

A **while** statement contains an expression that controls the repeated execution of a statement (which can be a compound statement).



The expression controlling the repetition must be of type *Boolean*. It is evaluated *before* the contained statement is executed. The contained statement is executed repeatedly as long as the expression is True. If the expression is False at the beginning, the statement is not executed at all.

Examples of **while** statements include:

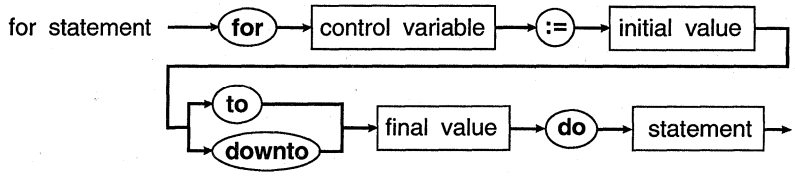
```
while Data[I] <> X do I := I + 1;
```

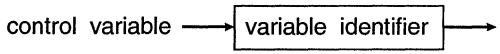
```
while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;
```

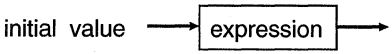
```
while not Eof(InFile) do
begin
  Readln(InFile, Line);
  Process(Line);
end;
```

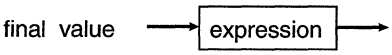
For statements

The **for** statement causes a statement (which can be a compound statement) to be repeatedly executed while a progression of values is assigned to a control variable.









The control variable must be a variable identifier (without any qualifier) that signifies a variable declared to be local to the block containing the **for** statement. The control variable must be of an ordinal type. The initial and final values must be of a type assignment-compatible with the ordinal type.

When a **for** statement is entered, the initial and final values are determined once for the remainder of the execution of the **for** statement.

The statement contained by the **for** statement is executed once for every value in the range *initial value* to *final value*. The control variable always starts off at *initial value*. When a **for** statement uses **to**, the value of the control variable is incremented by one for each repetition. If *initial value* is greater than *final value*, the contained statement is not executed. When a **for** statement uses **downto**, the value of the control variable is decremented by one for each repetition. If *initial value* value is less than *final value*, the contained statement is not executed.

It's an error if the contained statement alters the value of the control variable. After a **for** statement is executed, the value of the control variable value is undefined, unless execution of the **for** statement was interrupted by a **goto** from the **for** statement.

With these restrictions in mind, the **for** statement

```
for V := Expr1 to Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 <= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Succ(V);
      Body;
    end;
  end;
end;
```

and the **for** statement

```
for V := Expr1 downto Expr2 do Body;
```

is equivalent to

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 >= Temp2 then
  begin
    V := Temp1;
    Body;
    while V <> Temp2 do
    begin
      V := Pred(V);
      Body;
    end;
  end;
end;
```

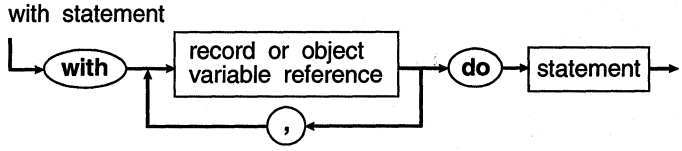
where *Temp1* and *Temp2* are auxiliary variables of the host type of the variable *V* and don't occur elsewhere in the program.

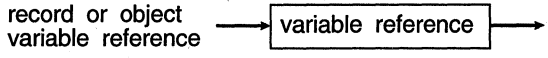
Examples of **for** statements follow:

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I]

for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Check(C);
```

With statements

The **with** statement is shorthand for referencing the fields of a record, and the fields, methods, constructor, and destructor of an object. Within a **with** statement, the fields of one or more specific record variables can be referenced using their field identifiers only. The syntax of a **with** statement follows:





Following is an example of a **with** statement:

```
with Date do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1
  end
  else
    Month := Month + 1;
```

This is equivalent to

```
if Date.Month = 12 then
begin
  Date.Month := 1;
  Date.Year := Date.Year + 1
end
else
  Date.Month := Date.Month + 1;
```

Within a **with** statement, each variable reference is first checked to see if it can be interpreted as a field of the record. If so, it is always interpreted as such, even if a variable with the same name is also accessible. Suppose the following declarations have been made:

```
type
  Point = record
    X, Y: Integer;
  end;
var
  X: Point;
  Y: Integer;
```

In this case, both *X* and *Y* can refer to a variable or to a field of the record. In the statement

```
with X do
begin
  X := 10;
  Y := 25;
end;
```

the *X* between **with** and **do** refers to the variable of type *Point*, but in the compound statement, *X* and *Y* refer to *X.X* and *X.Y*.

The statement

```
with V1, V2, ... Vn do S;
```

is equivalent to

```
with V1 do
  with V2 do
    ....
      with Vn do
        S;
```

In both cases, if *Vn* is a field of both *V1* and *V2*, it is interpreted as *V2.Vn*, not *V1.Vn*.

If the selection of a record variable involves indexing an array or dereferencing a pointer, these actions are executed once before the component statement is executed.

C H A P T E R

8

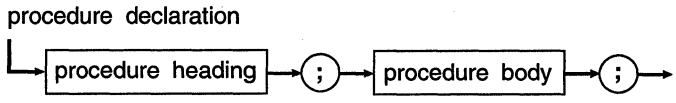
Procedures and functions

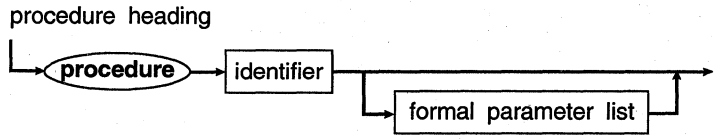
Procedures and functions allow you to nest additional blocks in the main program block. Each procedure or function declaration has a heading followed by a block. A procedure is activated by a procedure statement; a function is activated by the evaluation of an expression that contains its call and returns a value to that expression.

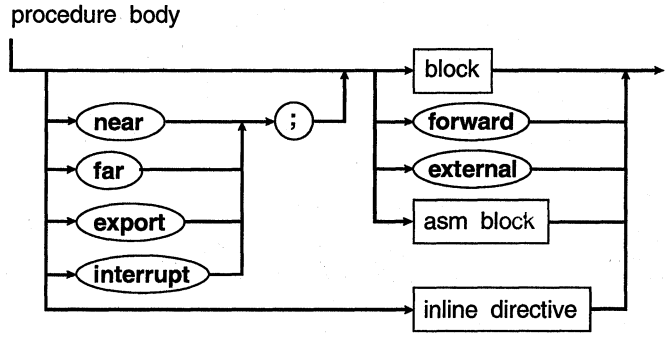
This chapter discusses the different types of procedure and function declarations and their parameters.

Procedure declarations

A procedure declaration associates an identifier with a block as a procedure; that procedure can then be activated by a procedure statement.







The syntax for a formal parameter list is shown in the section "Parameters" on page 108.

The procedure heading names the procedure's identifier and specifies the formal parameters (if any).

A procedure is activated by a procedure statement, which states the procedure's identifier and any actual parameters required. The statements to be executed on activation are noted in the statement part of the procedure's block. If the procedure's identifier is used in a procedure statement within the procedure's block, the procedure is executed recursively (it calls itself while executing).

Here's an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(N mod 10 + Ord('0')) + S;
    N := N div 10;
  until N = 0;
  if N < 0 then
    S := '-' + S;
end;
```

Near and far declarations

Near and far calls are described in full in Chapter 18, "Control issues."

Turbo Pascal supports two *procedure call models*: near and far. In terms of code size and execution speed, the near call model is the more efficient, but it carries the restriction that near procedures can only be called from within the module in which they are declared. Far procedures, on the other hand, can be called from any module, but the code for a far call is slightly less efficient.

Turbo Pascal will automatically select the correct call model based on a procedure's declaration: Procedures declared in the **interface** part of a unit use the far call model—they can be called from other modules. Procedures declared in a program or in the **implementation** part of a unit use the near call model—they can only be called from within that program or unit.

For some specific purposes, a procedure may be required to use the far call model. For example, in an overlaid application, all procedures and functions are generally required to be far; likewise, if a procedure or function is to be assigned to a procedural variable, it has to use the far call model. The **$F** compiler directive can be used to override the compiler's automatic call model selection. Procedures and functions compiled in the {**$F+**} state always use the far call model; in the {**$F-**} state, the compiler automatically selects the correct model. The default state is {**$F-**}.

To force a specific call model, a procedure declaration can optionally specify a **near** or **far** directive before the block—if such a directive is present, it overrides the setting of the **$F** compiler directive as well as the compiler's automatic call model selection.

Export declarations

The **export** directive makes a procedure or function "exportable" by forcing the routine to use the far call model and preparing the routine for export by generating special procedure entry and exit code.

Windows requires procedures and functions to be exportable in the following cases:

- Procedures and functions that are exported by a DLL (dynamic-link library)
- Callback procedures and functions

Chapter 10, "Dynamic-link libraries", discusses how to export procedures and functions in a DLL. Note that even though a procedure or function is compiled with an **export** directive, the actual exporting of the procedure or function doesn't occur until the routine is listed in a library's **exports** clause.

Callback procedures and functions are routines in your application that are called by Windows and not by your application itself. Callback routines must be compiled with the **export** directive, but they do not have to be listed in an **exports** clause. Some examples of common callback procedures and functions are:

- Window procedures
- Dialog procedures
- Enumeration callback procedures
- Memory-notification procedures
- Window-hook procedures (filters)

Interrupt declarations

Interrupt procedures are described in full in Chapter 18, "Control issues."

A procedure declaration can optionally specify an **interrupt** directive before the block, and the procedure is then considered an interrupt procedure. For now, note that interrupt procedures cannot be called from procedure statements, and that every interrupt procedure must specify a parameter list exactly like the following:

```
procedure MyInt(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES,
  BP: Word);
interrupt;
```

Instead of the block in a procedure or function declaration, you can write a **forward**, **external**, or **inline** declaration.

Forward declarations

A procedure declaration that specifies the directive **forward** instead of a block is a **forward** declaration. Somewhere after this declaration, the procedure must be defined by a *defining* declaration—a procedure declaration that uses the same procedure identifier but omits the formal parameter list and includes a block. The **forward** declaration and the defining declaration must appear in the same procedure and function declaration part. Other procedures and functions can be declared

between them, and they can call the forward-declared procedure. Mutual recursion is thus possible.

The **forward** declaration and the defining declaration constitute a complete procedure declaration. The procedure is considered declared at the **forward** declaration.

An example of a **forward** declaration follows:

```
procedure Walter(M, N: Integer); forward;

procedure Clara(X, Y: Real);
begin
  ...
  Walter(4, 5);
  ...
end;

procedure Walter;
begin
  ...
  Clara(8.3, 2.4);
  ...
end;
```

A procedure's defining declaration can be an **external** or **assembler** declaration; however, it cannot be a **near**, **far**, **export**, or **inline** declaration or another **forward** declaration. Likewise, the defining declaration cannot specify a **near**, **far**, **export**, or **interrupt** directive.

No **forward** declarations are allowed in the interface part of a unit.

External declarations

For further details on linking with assembly language, refer to Chapter 23.

External declarations allow you to interface with separately compiled procedures and functions written in assembly language. The **external** code must be linked with the Pascal program or unit through {**$L** *filename*} directives.

Examples of **external** procedure declarations follow:

```
procedure MoveWord(var Source, Dest; Count: Word); external;
procedure MoveLong(var Source, Dest; Count: Word); external;

procedure FillWord(var Dest; Data: Integer; Count: Word); external;
procedure FillLong(var Dest; Data: Longint; Count: Word); external;

{$L BLOCK.OBJ}
```

The **external** directive is also used to *import* procedures and functions from a DLL (dynamic-link library). For example, the following **external** declaration imports a function called *GlobalAlloc* from the DLL called 'KERNEL' (the Windows kernel).

```
function GlobalAlloc(Flags: Word; Bytes: Longint): THandle; far;
external 'KERNEL' index 15;
```

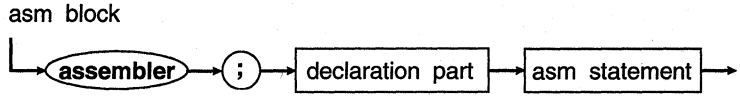
To read more about importing procedures and functions from a DLL, see Chapter 10, "Dynamic-link libraries".

The **external** directive takes the place of the declaration and statement parts in an imported procedure or function. Imported procedures and functions must use the far call model selected by using a **far** procedure directive or a {**$F+**} compiler directive. Aside from this requirement, imported procedures and functions are just like regular procedures and functions.

Assembler declarations

For further details on assembler procedures and functions, refer to Chapter 22, "The inline assembler."

Assembler declarations let you write entire procedures and functions in inline assembler.



Inline declarations

Inline procedures are described in full in Chapter 22, "The inline assembler."

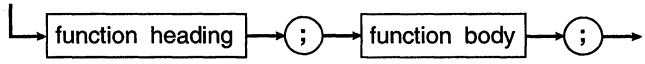
The **inline** directive permits you to write machine code instructions instead of the block. When a normal procedure is called, the compiler generates code that pushes the procedure's parameters onto the stack, and then generates a **CALL** instruction to call the procedure. When you "call" an **inline** procedure, the compiler generates code from the inline directive instead of the **CALL**. Thus, an **inline** procedure is "expanded" every time you refer to it, just like a macro in assembly language. Here's a short example of two **inline** procedures:

```
procedure DisableInterrupts; inline($FA);   { CLI }
procedure EnableInterrupts; inline($FB);    { STI }
```

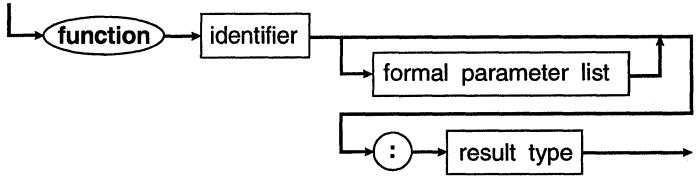
Function declarations

A **function** declaration defines a part of the program that computes and returns a value.

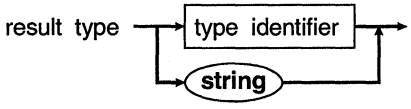
function declaration



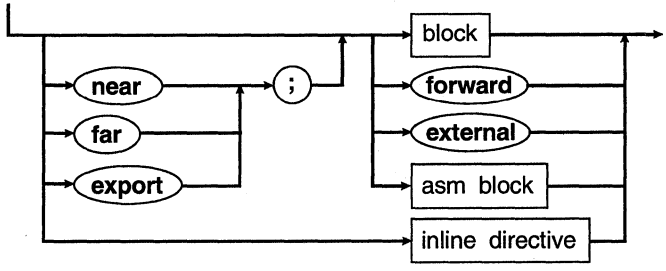
function heading



result type



function body



The **function** heading specifies the identifier for the function, the formal parameters (if any), and the function result type.

A function is activated by the evaluation of a **function** call. The **function** call gives the function's identifier and any actual parameters required by the function. A **function** call appears as an

operand in an expression. When the expression is evaluated, the function is executed, and the value of the operand becomes the value returned by the function.

The statement part of the function's block specifies the statements to be executed upon activation of the function. The block should contain at least one assignment statement that assigns a value to the function identifier. The result of the function is the last value assigned. If no such assignment statement exists or if it is not executed, the value returned by the function is unspecified.

If the function's identifier is used in a function call within the function's block, the function is executed recursively.

Following are examples of **function** declarations:

```
function Max(A: Vector; N: Integer): Extended;
var
  X: Extended;
  I: Integer;
begin
  X := A[1];
  for I := 2 to N do
    if X < A[I] then X := A[I];
  Max := X;
end;

function Power(X: Extended; Y: Integer): Extended;
var
  Z: Extended;
  I: Integer;
begin
  Z := 1.0; I := Y;
  while I > 0 do
  begin
    if Odd(I) then Z := Z * X;
    I := I div 2;
    X := Sqr(X);
  end;
  Power := Z;
end;
```

Like procedures, functions can be declared as **near**, **far**, **export**, **forward**, **external**, **assembler**, or **inline**; however, **interrupt** functions are *not* allowed.

Method declarations

The declaration of a method within an object type corresponds to a **forward** declaration of that method. Thus, somewhere after the object-type declaration and within the same scope as the object-type declaration, the method must be *implemented* by a defining declaration.

For procedure and function methods, the defining declaration takes the form of a normal procedure or function declaration, with the exception that the procedure or function identifier in this case is a qualified method identifier.

For constructor methods and destructor methods, the defining declaration takes the form of a procedure method declaration, except that the **procedure** reserved word is replaced by a **constructor** or **destructor** reserved word.

A method's defining declaration can optionally repeat the formal parameter list of the method heading in the object type. The defining declaration's method heading must in that case match exactly the order, types, and names of the parameters, and the type of the function result, if any.

In the defining declaration of a method, there is always an implicit parameter with the identifier *Self*, corresponding to a formal variable parameter that possesses the object type. Within the method block, *Self* represents the instance whose method component was designated to activate the method. Thus, any changes made to the values of the fields of *Self* are reflected in the instance.

The scope of a component identifier in an object type extends over any procedure, function, constructor, or destructor block that implements a method of the object type. The effect is the same as if the entire method block was embedded in a **with** statement of the form

```
with Self do begin ... end
```

For this reason, the spellings of component identifiers, formal method parameters, *Self*, and any identifiers introduced in a method implementation must be unique.

Here are some examples of method implementations:

```
procedure Rect.Intersect(var R: Rect);
begin
  if A.X < R.A.X then A.X := R.A.X;
  if A.Y < R.A.Y then A.Y := R.A.Y;
  if B.X > R.B.X then B.X := R.B.X;
  if B.Y > R.B.Y then B.Y := R.B.Y;
  if (A.X >= B.X) or (A.Y >= B.Y) then Init(0, 0, 0, 0);
end;

procedure Field.Display;
begin
  GotoXY(X, Y);
  Write(Name^, ' ', GetStr);
end;

function NumField.PutStr(S: String): Boolean;
var
  E: Integer;
begin
  Val(S, Value, E);
  PutStr := (E = 0) and (Value >= Min) and (Value <= Max);
end;
```

Constructors and destructors

Constructors and destructors are specialized forms of methods. Used in connection with the extended syntax of the *New* and *Dispose* standard procedures, constructors and destructors have the ability to allocate and deallocate dynamic objects. In addition, constructors have the ability to perform the required initialization of objects that contain virtual methods. Like other methods, constructors and destructors can be inherited, and an object can have any number of constructors and destructors.

Constructors are used to initialize newly instantiated objects. Typically, the initialization is based on values passed as parameters to the constructor. Constructors cannot be virtual, because the virtual method dispatch mechanism depends on a constructor first having initialized the object.

Here are some examples of constructors:

```
constructor Field.Copy(var F: Field);
begin
  Self := F;
end;
```

```
constructor Field.Init(FX, FY, FLen: Integer; FName: String);
begin
  X := FX;
  Y := FY;
  Len := FLen;
  GetMem(Name, Length(FName) + 1);
  Name^ := FName;
end;

constructor StrField.Init(FX, FY, FLen: Integer; FName: String);
begin
  Field.Init(FX, FY, FLen, FName);
  GetMem(Value, Len);
  Value^ := '';
end;
```

The first action of a constructor of a descendant type, such as the preceding *StrField.Init*, is almost always to call its immediate ancestor's corresponding constructor to initialize the inherited fields of the object. Having done that, the constructor then initializes the fields of the object that were introduced in the descendant.

Destructors can be virtual, and often are. Destructors seldom take any parameters.

Destructors are the counterparts of constructors, and are used to clean up objects after their use. Typically, the cleanup consists of disposing any pointer fields in the object.

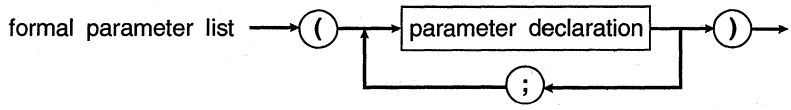
Here are some examples of destructors:

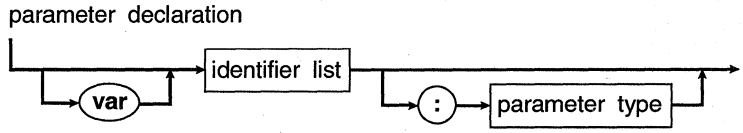
```
destructor Field.Done;
begin
  FreeMem(Name, Length(Name^) + 1);
end;

destructor StrField.Done;
begin
  FreeMem(Value, Len);
  Field.Done;
end;
```

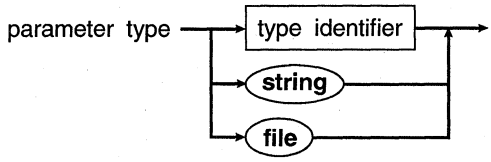
A destructor of a descendant type, such as the preceding *StrField.Done*, typically first disposes the pointer fields introduced in the descendant, and then, as its last action, calls the corresponding destructor of its immediate ancestor to dispose any inherited pointer fields of the object.

Parameters

The declaration of a procedure or function specifies a formal parameter list. Each parameter declared in a formal parameter list is local to the procedure or function being declared, and can be referred to by its identifier in the block associated with the procedure or function.







There are three kinds of parameters: *value, variable,* and *untyped variable.* They are characterized as follows:

- A parameter group without a preceding **var** and followed by a type is a list of value parameters.
- A parameter group preceded by **var** and followed by a type is a list of variable parameters.
- A parameter group preceded by **var** and not followed by a type is a list of untyped variable parameters.

Value parameters

A formal value parameter acts like a variable local to the procedure or function, except that it gets its initial value from the corresponding actual parameter upon activation of the procedure or function. Changes made to a formal value parameter do not affect the value of the actual parameter.

A value parameter's corresponding actual parameter in a procedure statement or function call must be an expression, and its value must not be of file type or of any structured type that contains a file type.

The actual parameter must be assignment-compatible with the type of the formal value parameter. If the parameter type is **string**, then the formal parameter is given a size attribute of 255.

Variable parameters

A variable parameter is employed when a value must be passed from a procedure or function to the caller. The corresponding actual parameter in a procedure statement or function call must be a variable reference. The formal variable parameter represents the actual variable during the activation of the procedure or function, so any changes to the value of the formal variable parameter are reflected in the actual parameter.

Within the procedure or function, any reference to the formal variable parameter accesses the actual parameter itself. The type of the actual parameter must be identical to the type of the formal variable parameter (you can bypass this restriction through untyped variable parameters). If the formal parameter type is **string**, it is given the length attribute 255, and the actual variable parameter must be a string type with a length attribute of 255.

File types can only be passed as variable parameters.

If referencing an actual variable parameter involves indexing an array or finding the object of a pointer, these actions are executed before the activation of the procedure or function.

Objects

The rules of object-type assignment compatibility also apply to object-type variable parameters: For a formal parameter of type *T1*, the actual parameter might be of type *T2* if *T2* is in the domain of *T1*. For example, the *Field.Copy* method might be passed an instance of *Field*, *StrField*, *NumField*, *ZipField*, or any other instance of a descendant of *Field*.

Untyped variable parameters

When a formal parameter is an untyped variable parameter, the corresponding actual parameter may be any variable reference, regardless of its type.

Within the procedure or function, the untyped variable parameter is typeless; that is, it is incompatible with variables of all other types, unless it is given a specific type through a variable typecast.

An example of untyped variable parameters follows:

```
function Equal(var Source, Dest; Size: Word): Boolean;
type
  Bytes = array[0..MaxInt] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (Bytes(Dest)[N] <> Bytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

This function can be used to compare any two variables of any size. For instance, given the declarations

```
type
  Vector = array[1..10] of Integer;
  Point  = record
    X, Y: Integer;
  end;
var
  Vec1, Vec2: Vector;
  N: Integer;
  P: Point;
```

then the function calls

```
Equal(Vec1, Vec2, SizeOf(Vector))
Equal(Vec1, Vec2, SizeOf(Integer) * N)
Equal(Vec[1], Vec1[6], SizeOf(Integer) * 5)
Equal(Vec1[1], P, 4)
```

compare *Vec1* to *Vec2*, compare the first *N* components of *Vec1* to the first *N* components of *Vec2*, compare the first five components of *Vec1* to the last five components of *Vec1*, and compare *Vec1[1]* to *P.X* and *Vec1[2]* to *P.Y*.

Procedural types

Procedural types are defined in Chapter 3, "Types."

As an extension to standard Pascal, Turbo Pascal allows procedures and functions to be treated as objects that can be assigned to variables and passed as parameters; *procedural types* make this possible.

Procedural variables

Once a procedural type has been defined, it becomes possible to declare variables of that type. Such variables are called *procedural variables*. For example, given the preceding type declarations, the following variables can be declared:

```
var
  P: SwapProc;
  F: MathFunc;
```

Like an integer variable that can be assigned an integer value, a procedural variable can be assigned a *procedural value*. Such a value can of course be another procedural variable, but it can also be a procedure or a function identifier. In this context, a procedure or function declaration can be viewed as a special kind of constant declaration, the value of the constant being the procedure or function. For example, given the following procedure and function declarations,

```
procedure Swap(var A, B: Integer); far;
var
  Temp: Integer;
begin
  Temp := A;
  A := B;
  B := Temp;
end;
```

```
function Tan(Angle: Real): Real; far;
begin
  Tan := Sin(Angle) / Cos(Angle);
end;
```

The variables *P* and *F* declared previously can now be assigned values:

```
P := Swap;
F := Tan;
```

Following these assignments, the call `P(I, J)` is equivalent to `Swap(I, J)`, and `F(X)` is equivalent to `Tan(X)`.

As in any other assignment operation, the variable on the left and the value on the right must be assignment-compatible. To be considered assignment-compatible, procedural types must have the same number of parameters, and parameters in corresponding positions must be of identical types; finally, the result types of functions must be identical. As mentioned previously, parameter names are of no significance when it comes to procedural-type compatibility.

In addition to being of a compatible type, a procedure or function must satisfy the following requirements if it is to be assigned to a procedural variable:

- It must be declared with a **far** directive or compiled in the {**$F+**} state.
- It cannot be
  - a standard procedure or function
  - a nested procedure or function
  - an **inline** procedure or function
  - an **interrupt** procedure or function

Standard procedures and functions are the procedures and functions declared by the *System* unit, such as *Writeln, Readln, Chr,* and *Ord*. To use a standard procedure or function with a procedural variable, you will have to write a "shell" around it. For example, given the procedural type

```
type
  IntProc = procedure(N: Integer);
```

the following is an assignment-compatible procedure to write an integer:

```
procedure WriteInt(Number: Integer); far;
begin
  Write(Number);
end;
```

Nested procedures and function cannot be used with procedural variables. A procedure or function is nested when it is declared within another procedure or function. In the following example, *Inner* is nested within *Outer*, and *Inner* cannot therefore be assigned to a procedural variable.

```
program Nested;
procedure Outer;
procedure Inner;
begin
  Writeln('Inner is nested');
end;
begin
  Inner;
end;
begin
  Outer;
end.
```

The use of procedural types is not restricted to simple procedural variables. Like any other type, a procedural type can participate in the declaration of a structured type, as demonstrated by the following declarations:

```
type
  GotoProc  = procedure(X, Y: Integer);
  ProcList  = array[1..10] of GotoProc;
  WindowPtr = ^WindowRec;
  WindowRec = record
    Next: WindowPtr;
    Header: string[31];
    Top, Left, Bottom, Right: Integer;
    SetCursor: GotoProc;
  end;
var
  P: ProcList;
  W: WindowPtr;
```

Given the preceding declarations, the following statements are valid procedure calls:

```
P[3](1, 1);
W^.SetCursor(10, 10);
```

When a procedural value is assigned to a procedural variable, what physically takes place is that the address of the procedure is stored in the variable. In fact, a procedural variable is much like a pointer variable, except that instead of pointing to data, it points to a procedure or function. Like a pointer, a procedural variable occupies 4 bytes (two words), containing a memory address. The first word stores the offset part of the address, and the second word stores the segment part.

Procedural-type parameters

Since procedural types are allowed in any context, it is possible to declare procedures or functions that take procedures or functions as parameters. The following program demonstrates the use of a procedural-type parameter to output three tables of different arithmetic functions:

```
program Tables;

type
  Func = function(X, Y: Integer): Integer;

function Add(X, Y: Integer): Integer; far;
begin
  Add := X + Y;
end;

function Multiply(X, Y: Integer): Integer; far;
begin
  Multiply := X * Y;
end;

function Funny(X, Y: Integer): Integer; far;
begin
  Funny := (X + Y) * (X - Y);
end;

procedure PrintTable(W, H: Integer; Operation: Func);
var
  X, Y: Integer;
begin
  for Y := 1 to H do
  begin
    for X := 1 to W do
      Write(Operation(X, Y):5);
    Writeln;
  end;
  Writeln;
end;
```

```
begin
  PrintTable(10, 10, Add);
  PrintTable(10, 10, Multiply);
  PrintTable(10, 10, Funny);
end.
```

When run, the *Tables* program outputs three tables. The second one looks like this:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Procedural-type parameters are particularly useful in situations where a certain common action is to be carried out on a set of procedures or functions. In this case, the *PrintTable* procedure represents the common action to be carried out on the *Add*, *Multiply*, and *Funny* functions.

If a procedure or function is to be passed as a parameter, it must conform to the same type-compatibility rules as in an assignment. Thus, such procedures and functions must be declared with a **far** directive, they cannot be built-in routines, they cannot be nested, and they cannot be declared with the **inline** or **interrupt** attributes.

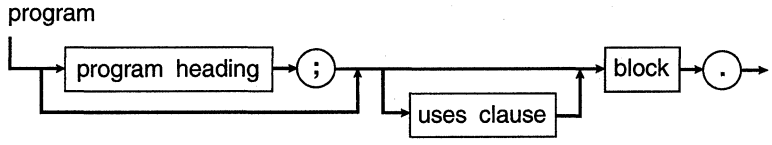
C H A P T E R

9

Programs and units

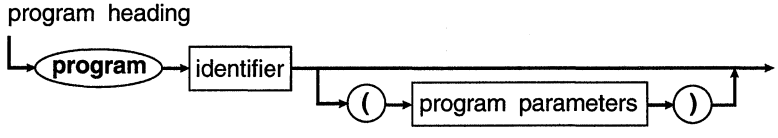
Program syntax

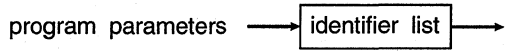
A Turbo Pascal program takes the form of a procedure declaration except for its heading and an optional **uses** clause.



The program heading

The program heading specifies the program's name and its parameters.

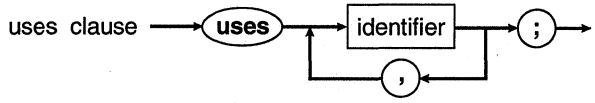




The program heading, if present, is purely decorative and is ignored by the compiler.

The uses clause

The **uses** clause identifies all units used by the program, including units used directly and units used by those units.



The *System* unit is always used automatically. *System* implements all low-level, run-time support routines to support such features as file I/O, string handling, floating point, dynamic memory allocation, and others.

Apart from *System*, Turbo Pascal implements many standard units, such as *WinDos*, *WinTypes*, and *WinProcs*. These are not used automatically; you must include them in your **uses** clause, for instance,

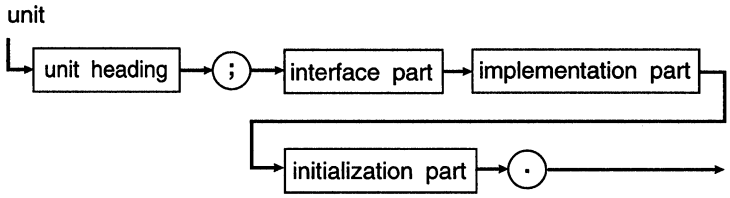
```
uses WinTypes, WinProcs;          { Can now use WinTypes and WinProcs }
```

See "The initialization part" on page 121.

The order of the units listed in the **uses** clause determines the order of their initialization.

Unit syntax

Units are the basis of modular programming in Turbo Pascal. They are used to create libraries that you can include in various programs without making the source code available, and to divide large programs into logically related modules.



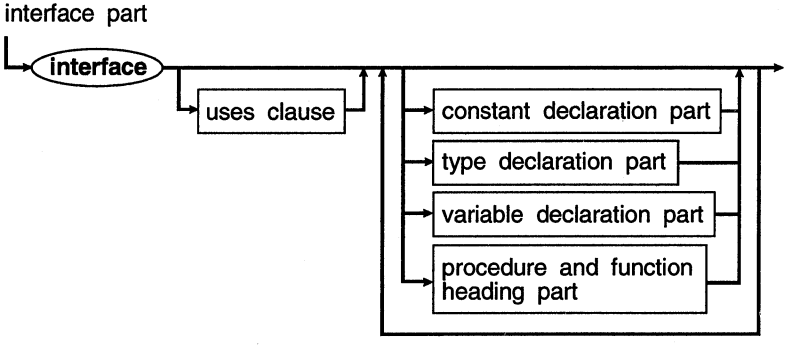
The unit heading

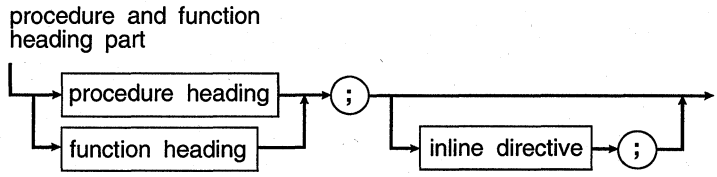
The unit heading specifies the unit's name.

The unit name is used when referring to the unit in a **uses** clause. The name must be unique—two units with the same name cannot be used at the same time.

The interface part

The interface part declares constants, types, variables, procedures, and functions that are *public*, that is, available to the host (the program or unit using the unit). The host can access these entities as if they were declared in a block that encloses the host.

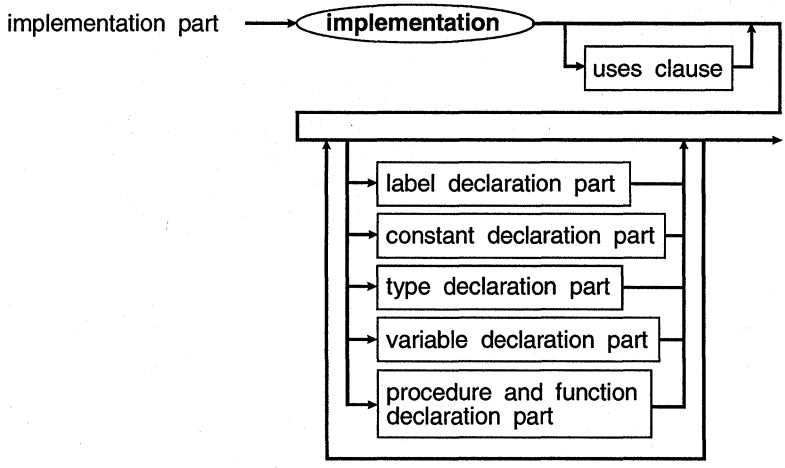




Unless a procedure or function is **inline**, the interface part only lists the procedure or function heading. The block of the procedure or function follows in the implementation part.

The implementation part

The implementation part defines the block of all public procedures and functions. In addition, it declares constants, types, variables, procedures, and functions that are *private*, that is, not available to the host.

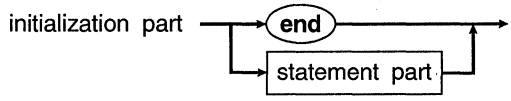


In effect, the procedure and function declarations in the interface part are like **forward** declarations, although the **forward** directive is not specified. Therefore, these procedures and functions can be defined and referenced in any sequence in the implementation part.

Procedure and function headings can be duplicated from the interface part. You don't have to specify the formal parameter list, but if you do, the compiler will issue a compile-time error if the interface and implementation declarations don't match.

The initialization part

The initialization part is the last part of a unit. It consists either of the reserved word **end** (in which case the unit has no initialization code) or of a statement part to be executed in order to initialize the unit.



The initialization parts of units used by a program are executed in the same order that the units appear in the **uses** clause.

Indirect unit references

The **uses** clause in a module (program or unit) need only name the units used directly by that module. Consider the following example:

```
program Prog;
uses Unit2;
const a = b;
begin
end.

unit Unit2;
interface
uses Unit1;
const b = c;
implementation
end.

unit Unit1;
interface
const c = 1;
implementation
const d = 2;
end.
```

In the previous example, *Unit2* is directly dependent on *Unit1*, and *Prog* is directly dependent on *Unit2*. Furthermore, *Prog* is indirectly dependent on *Unit1* (through *Unit2*), even though none of the identifiers declared in *Unit1* are available to *Prog*.

In order to compile a module, Turbo Pascal must be able to locate all units upon which the module depends (directly or indirectly). So, to compile *Prog* above, the compiler must be able to locate both *Unit1* and *Unit2*, or else an error occurs.

When changes are made in the interface part of a unit, other units using the unit must be recompiled. However, if changes are only made to the implementation or the initialization part, other units that use the unit *need not* be recompiled. In the previous example, if the interface part of *Unit1* is changed (for example, $c = 2$) *Unit2* must be recompiled; changing the implementation part (for example, $d = 1$) doesn't require recompilation of *Unit2*.

When a unit is compiled, Turbo Pascal computes a *unit version number*, which is basically a checksum of the interface part. In the preceding example, when *Unit2* is compiled, the current version number of *Unit1* is saved in the compiled version of *Unit2*. When *Prog* is compiled, the version number of *Unit1* is checked against the version number stored in *Unit2*. If the version numbers do not match, indicating that a change was made in the interface part of *Unit1* since *Unit2* was compiled, the compiler shows an error or recompiles *Unit2*, depending on the mode of compilation.

Circular unit references

Placing a **uses** clause in the implementation section of a unit allows you to further hide the inner details of the unit, since units used in the implementation section are not visible to users of the unit. More importantly, however, it also enables you to construct mutually dependent units.

The following program demonstrates how two units can "use" each other. The main program, *Circular*, uses a unit named *Display*. *Display* contains one routine in its interface section, *WriteXY*, which takes three parameters: an (x, y) coordinate pair and a text message to display. If the (x, y) coordinates are onscreen, *WriteXY* moves the cursor to (x, y) and displays the message there; otherwise, it calls a simple error-handling routine.

So far, there's nothing fancy here—*WriteXY* is taking the place of *Write*. Here's where the circular unit reference enters in: How is

the error-handling routine going to display its error message? By using *WriteXY* again. Thus you have *WriteXY*, which calls the error-handling routine *ShowError*, which in turn calls *WriteXY* to put an error message onscreen. If your head is spinning in circles, let's look at the source code to this example, so you can see that it's really not that tricky.

The main program, *Circular*, clears the screen and makes three calls to *WriteXY*:

```
program Circular;
{ Display text using WriteXY }

uses
  WinCrt, Display;

begin
  ClrScr;
  WriteXY(1, 1, 'Upper left corner of screen');
  WriteXY(100, 100, 'Way off the screen');
  WriteXY(81 - Length('Back to reality'), 15, 'Back to reality');
end.
```

Look at the (*x*, *y*) coordinates of the second call to *WriteXY*. It's hard to display text at (100, 100) on an 80x25 line screen. Next, let's see how *WriteXY* works. Here's the source to the *Display* unit, which contains the *WriteXY* procedure. If the (*x*, *y*) coordinates are valid, it displays the message; otherwise, *WriteXY* displays an error message:

```
unit Display;
{ Contains a simple video display routine }

interface

procedure WriteXY(X, Y: Integer; Message: String);

implementation
uses
  WinCrt, Error;

procedure WriteXY(X, Y: Integer; Message: String);
begin
  if (X in [1..80]) and (Y in [1..25]) then
  begin
    GoToXY(X, Y);
    Write(Message);
  end;
```

```
  else
    ShowError('Invalid WriteXY coordinates');
end;

end.
```

The *ShowError* procedure called by *WriteXY* is declared in the following code in the *Error* unit. *ShowError* always displays its error message on the 25th line of the screen:

```
unit Error;
{ Contains a simple error-reporting routine }

interface

procedure ShowError(ErrMsg: String);

implementation

uses
  Display;

procedure ShowError(ErrMsg: String);
begin
  WriteXY(1, 25, 'Error: ' + ErrMsg);
end;

end.
```

Notice that the **uses** clause in the **implementation** sections of both *Display* and *Error* refer to each other. These two units can refer to each other in their **implementation** sections because Turbo Pascal can compile complete **interface** sections for both. In other words, the Turbo Pascal compiler will accept a reference to partially compiled unit *A* in the **implementation** section of unit *B*, as long as both *A* and *B*'s **interface** sections do not depend upon each other (and thus follow Pascal's strict rules for declaration order).

Sharing other declarations

What if you want to modify *WriteXY* and *ShowError* to take an additional parameter that specifies a rectangular window onscreen:

```
procedure WriteXY(SomeWindow: WindRec; X, Y: Integer;
  Message: String);

procedure ShowError(SomeWindow: WindRec; ErrMsg: String);
```

Remember these two procedures are in separate units. Even if you declared *WindData* in the **interface** of one, there would be no legal way to make that declaration available to the **interface** of the other. The solution is to declare a third module that contains only the definition of the window record:

```
unit WindData;
interface
type
  WinDRec = record
    X1, Y1, X2, Y2: Integer;
    ForeColor, BackColor: Byte;
    Active: Boolean;
  end;
implementation
end.
```

In addition to modifying the code to *WriteXY* and *ShowError* to make use of the new parameter, the **interface** sections of both the *Display* and *Error* units can now "use" *WindData*. This approach is legal because unit *WindData* has no dependencies in its **uses** clause, and units *Display* and *Error* refer to each other only in their respective **implementation** sections.

C H A P T E R

10

Dynamic-link libraries

In the Windows environment, dynamic-link libraries (DLLs) permit several applications to share code and resources. With Turbo Pascal for Windows you can use DLLs as well as write your own DLLs to be used by other applications.

What is a DLL?

A DLL is an executable module containing code or resources for use by other applications or DLLs. In the traditional Turbo Pascal world, the concept closest to a DLL is a unit—both have the ability to provide services, in the form of procedures and functions, to a program. There are, however, many differences between DLLs and units. In particular, units are *statically linked*, whereas DLLs are *dynamically linked*.

When a program uses a procedure or function from a unit, a *copy* of that procedure or function's code is statically linked into the program's executable file. If two programs are running simultaneously and they use the same procedure or function from a unit, there will be two copies of that routine present in the system. It would be more efficient if the two programs could *share* a single copy of the routine; DLLs provide that ability.

In contrast to a unit, the code in a DLL is *not* linked into a program that uses the DLL. Instead, a DLL's code and resources are in a separate executable file with a .DLL extension. This file

must be present when the client program runs. The Windows program loader dynamically links the procedure and function calls in the program to their entry points in the DLLs used by the application.

A DLL doesn't have to be written in Turbo Pascal for a Turbo Pascal application to be able to use it. Also, programs written in other languages can use DLLs written in Turbo Pascal. DLLs are therefore ideal for multi-language programming projects.

Using DLLs

For a module to use a procedure or function in a DLL, the module must *import* the procedure or function using an **external** declaration. For example, the following **external** declaration imports a function called *GlobalAlloc* from the DLL called 'KERNEL' (the Windows kernel).

```
function GlobalAlloc(Flags: Word; Bytes: Longint): THandle; far;
external 'KERNEL' index 15;
```

In imported procedures and functions, the **external** directive takes the place of the declaration and statement parts that would otherwise be present. Imported procedures and functions must use the far call model selected by using a **far** procedure directive or a {**$F+**} compiler directive, but otherwise they behave no differently than normal procedures and functions.

Turbo Pascal for Windows has three ways to import procedures and functions:

- by name
- by new name
- by ordinal

The format of **external** directives for each of the three methods is demonstrated in the following example.

When no **index** or **name** clause is specified, the procedure or function is imported by name. The name used is the same as the procedure or function's identifier. In this example, the *ImportByName* procedure is imported from 'TESTLIB' using the name 'IMPORTBYNAME':

```
procedure ImportByName;    external 'TESTLIB';
```

When a **name** clause is specified, the procedure or function is imported by a different name than its identifier. Here the *ImportByNewName* procedure is imported from 'TESTLIB' using the name 'REALNAME':

```
procedure ImportByNewName; external 'TESTLIB' name 'REALNAME';
```

Finally, when an **index** clause is present, the procedure or function is imported by ordinal. Importing by ordinal reduces the load time of the module since Windows doesn't have to look up the name in the DLL's name table. In the example, the *ImportByOrdinal* procedure is imported as the fifth entry point in the DLL called 'TESTLIB':

```
procedure ImportByOrdinal; external 'TESTLIB' index 5;
```

The DLL name specified after the **external** keyword and the new name specified in a **name** clause don't have to be string literals. Any constant string expression is allowed. Likewise, the ordinal number specified in an **index** clause can be any constant integer expression.

```
const
  TestLib = 'TESTLIB';
  Ordinal = 5;

procedure ImportByName;    external TestLib;
procedure ImportByNewName; external TestLib name 'REALNAME';
procedure ImportByOrdinal; external TestLib index Ordinal;
```

Although a DLL can have variables, it is not possible to import them in other modules. Any access to a DLL's variables must take place through a procedural interface.

Import units

Declarations of imported procedures and functions can be placed directly in the program that imports them. Usually, though, they are grouped together in an "import unit" that contains declarations for all procedures and functions in a DLL, along with any constants and types required to interface with the DLL. The *WinTypes* and *WinProcs* units supplied with Turbo Pascal for Windows are examples of such import units. Import units are *not* a requirement of the DLL interface, but they do simplify maintenance of projects that use multiple DLLs.

As an example, consider a DLL called DATETIME.DLL that has four routines to get and set the date and time, using a record type

that contains the day, month, and year, and another record type that contains the second, minute, and hour. Instead of specifying the associated procedure, function, and type declarations in every program that uses the DLL, you can construct an import unit to go along with the DLL.

```
unit DateTime;

interface

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

type
  TDateRec = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

procedure SetTime(var Time: TTimeRec);
procedure GetTime(var Time: TTimeRec);
procedure SetDate(var Date: TDateRec);
procedure GetDate(var Date: TDateRec);

implementation

procedure SetTime; external 'DATETIME' index 1;
procedure GetTime; external 'DATETIME' index 2;
procedure SetDate; external 'DATETIME' index 3;
procedure GetDate; external 'DATETIME' index 4;

end.
```

Any program that uses DATETIME.DLL can now simply specify *DateTime* in its **uses** clause.

```
program ShowTime;

uses WinCrt, DateTime;

var
  Time: TTimeRec;

begin
  GetTime(Time);
  with Time do
    WriteLn('The time is ', Hour, ':', Minute, ':', Second);
end.
```

Another advantage of an import unit such as *DateTime* is that when the associated DATETIME.DLL is modified, only one unit, the *DateTime* import unit, needs updating to reflect the changes.

Static and dynamic imports

The **external** directive provides the ability to *statically* import procedures and functions from a DLL. A statically imported procedure or function always refers to the same entry point in the same DLL. Windows also supports *dynamic* imports, whereby the DLL name and the name or ordinal number of the imported procedure or function is specified at run time. The *ShowTime* program shown here uses *dynamic* importing to call the *GetTime* procedure in DATETIME.DLL. Note the use of a procedural type variable to represent the address of the *GetTime* procedure.

```
program ShowTime;

uses WinProcs, WinTypes, WinCrt;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;

begin
  Handle := LoadLibrary('DATETIME.DLL');
  if Handle >= 32 then
  begin
    GetTime := TGetTime(GetProcAddress(Handle, 'GETTIME'));
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

The structure of a Turbo Pascal DLL is identical to that of a program, except that a DLL starts with a **library** header instead of a **program** header. The **library** header tells Turbo Pascal to produce an executable file with the extension .DLL instead of .EXE, and also ensures that the executable file is marked as being a DLL.

The example here implements a very simple DLL with two exported functions, *Min* and *Max*, that calculate the smaller and larger of two integer values.

```
library MinMax;

function Min(X, Y: Integer): Integer; export;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; export;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min index 1,
  Max index 2;

begin
end.
```

Note the use of the **export** procedure directive to prepare *Min* and *Max* for exporting, and the **exports** clause to actually export the two routines, supplying an optional ordinal number for each of them.

Although the preceding small example doesn't demonstrate it, libraries can and often do consist of several units. In such cases, the library source file itself is frequently reduced to a **uses** clause, an **exports** clause, and the library's initialization code. For example,

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors index 1,
  DoneEditors index 2,
```

```
    InsertText index 3,
    DeleteSelection index 4,
    FormatSelection index 5,
    PrintSelection index 6,
    .
    .
    SetErrorHandler index 53;

begin
  InitLibrary;
end.
```

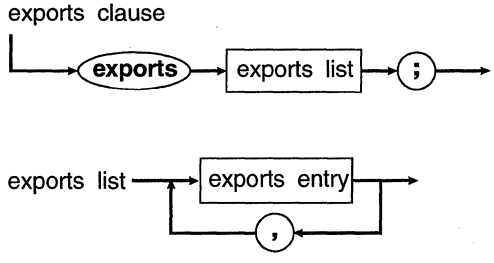
The export procedure directive

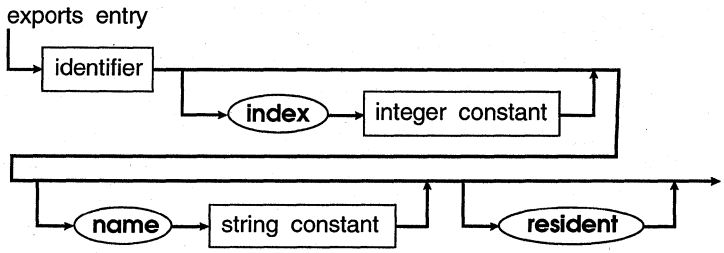
If procedures and functions are to be exported by a DLL, they must be compiled with the **export** procedure directive. The **export** directive belongs to the same family of procedure directives as the **near**, **far**, **inline**, and **interrupt** directives. This means that an **export** directive, if present, must be specified upon the first introduction of procedure or function—it cannot be supplied in the defining declaration of a **forward** declaration.

The **export** directive makes a procedure or function "exportable." It forces the routine to use the far call model and prepares the routine for export by generating special procedure entry and exit code. Note, however, that the actual exporting of the procedure or function doesn't occur until the routine is listed in a library's **exports** clause.

The exports clause

A procedure or function is exported by a DLL when it is listed in the library's **exports** clause.





An **exports** clause can appear anywhere and any number of times in a program or library's declaration part. Each entry in an **exports** clause specifies the identifier of a procedure or function to be exported. The identifier must denote a procedure or function that was compiled with the **export** directive. The identifier may be fully qualified if necessary; that is, you can precede it with a unit identifier and a period.

An exports entry can also include an **index** clause, which consists of the word **index** followed by an integer constant between 1 and 32,767. When an **index** clause is specified, the procedure or function to be exported uses the specified ordinal number. If no **index** clause is present in an exports entry, an ordinal number is automatically assigned.

An entry can also have a **name** clause, which consists of the word **name** followed by a string constant. When there is a **name** clause, the procedure or function to be exported uses the name specified by the string constant. If no **name** clause is present in an exports entry, the procedure or function is exported by its identifier and converted to all uppercase.

Finally, an exports entry can include the **resident** keyword. When **resident** is specified, the export information stays in memory while the DLL is loaded. The **resident** option significantly reduces the time it takes Windows to look up a DLL entry by name, so if client programs that use the DLL are likely to import certain entries by name, those entries should be exported using the **resident** keyword.

A program may contain an **exports** clause, but it seldom does because Windows doesn't allow application modules to export functions for use by other applications.

Library initialization and exit code

The statement part of a library constitutes the library's *initialization code*. The initialization code is executed once, when the library is initially loaded. When subsequent applications that use the library are loaded, Windows doesn't execute the initialization code again, but simply increments the DLL's use count.

A DLL is kept in memory as long as its use count is greater than zero. When the use count becomes zero, indicating that all applications that used the DLL have terminated, the DLL is removed from memory. At that point, the library's exit procedures are executed. Exit procedures are registered using the *ExitProc* variable, as described in Chapter 18, "Control issues."

A DLL's initialization code typically performs tasks such as registering window classes for window procedures contained in the DLL and setting initial values for the DLL's global variables. The initialization code of a library can signal an error condition by setting the *ExitCode* variable to zero (*ExitCode* is declared by the *System* unit). *ExitCode* defaults to 1, indicating initialization was successful. If the initialization code sets *ExitCode* to zero, the DLL is unloaded from system memory.

When a library's exit procedures are executed, the *ExitCode* variable does not contain a process termination code, as is the case with a program. Instead, *ExitCode* contains one of the values *wep_System_Exit* or *wep_Free_DLL*, which are defined in the *WinTypes* unit. *wep_System_Exit* indicates that Windows is shutting down, whereas *wep_Free_DLL* indicates that just this single DLL is being unloaded.

Here is an example of a library with initialization code and an exit procedure:

```
library Test;

uses WinTypes, WinProcs;

var
  SaveExit: Pointer;

procedure LibExit; far;
begin
  if ExitCode = wep_System_Exit then
  begin
    .
    .
```

```
    { System shutdown in progress }
    .
    .
  end else
  begin
    .
    .
    { DLL is being unloaded }
    .
    .
  end;
  ExitProc := SaveExit;
end;

begin
  .
  .
  { Perform DLL initialization }
  .
  .
  SaveExit := ExitProc;     { Save old exit procedure pointer }
  ExitProc := @LibExit;     { Install LibExit exit procedure }
end.
```

When Windows unloads a DLL, it first looks for an exported function called WEP in the DLL, and calls it if it is present. A Turbo Pascal library automatically exports a WEP function, which simply keeps calling the address stored in the *ExitProc* variable until *ExitProc* becomes **nil**. Since this works the way exit procedures are handled in Turbo Pascal programs, you can use the same exit procedure logic in both programs and libraries.

Exit procedures in a DLL *must* be compiled with stack-checking disabled (the {**$S-**} state), since Windows switches to an internal stack when terminating a DLL. Also, Windows will crash if a run-time error occurs in a DLL exit procedure, so you must include sufficient checks in your code to prevent run-time errors.

Library programming notes

The remaining sections present important facts you should keep in mind while working with DLLs.

Global variables in a DLL

A DLL has its own data segment, and any variables declared in a DLL are strictly private to that DLL. A DLL cannot access variables declared by modules that call the DLL, and likewise it is not possible for a DLL to export its variables for use by other modules. Such access must take place through a procedural interface.

Global memory and files in a DLL

As a rule, a DLL doesn't "own" any files that it opens or any global memory blocks that it allocates from the system. Such objects are owned by the application that (directly or indirectly) called the DLL. When an application terminates, any open files owned by it are automatically closed, and any global memory blocks owned by it are automatically deallocated. This means that file and global memory block handles stored in global variables in a DLL can become invalid at any time without the DLL being notified. For that reason, DLLs should refrain from making assumptions about the validity of file and global memory block handles stored in global variables across calls to the DLL. Instead, such object handles should be made parameters of the procedures and functions of the DLL and the calling application should be responsible for maintaining them.

DLLs and the System unit

During a DLL's lifetime, the *HInstance* variable contains the instance handle of the DLL. The *HPrevInst* and *CmdShow* variables are always zero in a DLL, as is the *PrefixSeg* variable since a DLL does not have a Program Segment Prefix (PSP). *PrefixSeg* is never zero in an application, so the test *PrefixSeg* <> 0 will return *True* if the current module is an application and *False* if the current module is a DLL.

For more details about the heap manager, refer to Chapter 16, "Memory issues."

In order to ensure proper operation of the heap manager contained in the *System* unit, the start-up code of a library sets the *HeapLimit* variable to zero. This allocates a unique global memory block with each call to *New* and *GetMem*, in effect disabling the heap manager's suballocation ability. Since each global memory block carries an overhead of at least 32 bytes, DLLs should avoid allocating large numbers of small heap blocks. If you can

guarantee that a DLL is used *only* by one application at a time, you can re-enable the suballocation feature by storing the normal value of 1024 in the *HeapLimit* variable.

Run-time errors in DLLs

If a run-time error occurs in a DLL, the *application* that called the DLL terminates. The DLL itself is not necessarily removed from memory at that time because other applications might still be using it. Since a DLL has no way of knowing whether it was called from a Turbo Pascal application or an application written in another programming language, it is not possible for the DLL to cause invocation of the application's exit procedures before the application is terminated. The application is simply aborted and removed from memory. For this reason, make sure there are sufficient checks in any DLL code so such errors don't occur.

DLLs and stack segments

Unlike an application, a DLL does not have its own stack segment. Instead, it uses the stack segment of the application that called the DLL. This can create problems in DLL routines that assume that the DS and SS registers refer to the same segment, which is the case in an application module. The Turbo Pascal compiler never generates code that assumes DS = SS, and none of the Turbo Pascal run-time library routines make this assumption. If you write assembly language code, however, be sure you don't assume that SS and DS registers contain the same value.

PART 2

The standard libraries

C H A P T E R

11

The System unit

The *System* unit is Turbo Pascal's run-time library. It implements low-level, run-time support routines for all built-in features, such as file I/O, string handling, floating point, and dynamic memory allocation. The *System* unit is used automatically by any unit or program and doesn't need to be referred to in a **uses** clause.

Standard procedures and functions

For more detailed information, refer to Chapter 24 in the Windows Programming Guide.

This section briefly describes all the standard (built-in) procedures and functions in Turbo Pascal, except for the I/O procedures and functions discussed in the next section (see page 145).

Standard procedures and functions are predeclared. Since all predeclared entities act as if they were declared in a block surrounding the program, no conflict arises from a declaration that redefines the same identifier within the program.

Flow control procedures

Procedure	Description
Exit	Exits immediately from the current block.
Halt	Stops program execution and returns to the operating system.
RunError	Stops program execution and generates a run-time error.

Dynamic allocation procedures

The dynamic allocation procedures and functions are used to manage the heap—a memory area that occupies all or some of the free memory left when a program is executed. Heap management techniques are discussed in the section "The heap manager" of Chapter 16.

Procedure	Description
Dispose	Disposes a dynamic variable.
FreeMem	Disposes a dynamic variable of a given size.
GetMem	Creates a new dynamic variable of a given size and sets a pointer variable to point to it.
New	Creates a new dynamic variable and sets a pointer variable to point to it.

Dynamic allocation functions

Function	Description
MaxAvail	Returns the size of the largest contiguous free block in the heap, indicating the size of the largest dynamic variable that can be allocated at the time of the call to *MaxAvail*.
MemAvail	Returns the number of free bytes of heap storage available.

Transfer functions

The transfer procedures *Pack* and *Unpack*, as defined in standard Pascal, are not implemented by Turbo Pascal.

Function	Description
Chr	Returns a character of a specified ordinal number.
Ord	Returns the ordinal number of an ordinal-type value.
Round	Rounds a type Real value to a type Longint value.
Trunc	Truncates a type Real value to a type Longint value.

Arithmetic functions

*When you're compiling in numeric processing mode, {**$N+**}, the return values of the floating-point routines in the System unit (Sqrt, Pi, Sin, and so on) are of type Extended instead of Real.*

Function	Description
Abs	Returns the absolute value of the argument.
ArcTan	Returns the arctangent of the argument.
Cos	Returns the cosine of the argument.
Exp	Returns the exponential of the argument.
Frac	Returns the fractional part of the argument.
Int	Returns the integer part of the argument.
Ln	Returns the natural logarithm of the argument.
Pi	Returns the value of *Pi* (3.1415926535897932385).
Sin	Returns the sine of the argument.
Sqr	Returns the square of the argument.
Sqrt	Returns the square root of the argument.

Ordinal procedures

Procedure	Description
Dec	Decrements a variable.
Inc	Increments a variable.

Ordinal functions

Function	Description
Odd	Tests if the argument is an odd number.
Pred	Returns the predecessor of the argument.
Succ	Returns the successor of the argument.

String procedures

Procedure	Description
Delete	Deletes a substring from a string.
Insert	Inserts a substring into a string.
Str	Converts a numeric value to its string representation.
Val	Converts the string value to its numeric representation.

String functions

Function	Description
Concat	Concatenates a sequence of strings.
Copy	Returns a substring of a string.
Length	Returns the dynamic length of a string.
Pos	Searches for a substring in a string.

Pointer and address functions

Function	Description
Addr	Returns the address of a specified object.
CSeg	Returns the current value of the CS register.
DSeg	Returns the current value of the DS register.
Ofs	Returns the offset of a specified object.
Ptr	Converts a segment base and an offset address to a pointer-type value.
Seg	Returns the segment of a specified object.
SPtr	Returns the current value of the SP register.
SSeg	Returns the current value of the SS register.

Miscellaneous functions

Function	Description
Hi	Returns the high-order byte of the argument.
Lo	Returns the low-order byte of the argument.
ParamCount	Returns the number of parameters passed to the program on the command line.
ParamStr	Returns a specified command-line parameter.
Random	Returns a random number.
SizeOf	Returns the number of bytes occupied by the argument.
Swap	Swaps the high- and low-order bytes of the argument.
UpCase	Converts a character to uppercase.

Miscellaneous procedures

Procedure	Description
FillChar	Fills a specified number of contiguous bytes with a specified value.
Move	Copies a specified number of contiguous bytes from a source range to a destination range.
Randomize	Initializes the built-in random generator with a random value.

File input and output

This section briefly describes the standard (or built-in) input and output (I/O) procedures and functions of Turbo Pascal. For more detailed information, refer to Chapter 19.

An introduction to file I/O

A Pascal file variable is any variable whose type is a file type. There are three classes of Pascal files: *typed*, *text*, and *untyped*.

The syntax for writing file types is given in the section "Structured types" in Chapter 3.

Before a file variable can be used, it must be associated with an external file through a call to the *Assign* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be "opened" to prepare it for input or output. An existing file can be opened via the *Reset* procedure and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. Each component has a component number. The first component of a file is considered to be component zero.

Files are normally accessed *sequentially*; that is, when a component is read using the standard procedure *Read* or written using the

standard procedure *Write*, the current file position moves to the next numerically-ordered file component. However, typed files and untyped files can also be accessed randomly via the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure *Close*. After closing a file completely, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors: If an error occurs, the program terminates, displaying a run-time error message. This automatic checking can be turned on and off using the {**$I+**} and {**$I-**} compiler directives. When I/O checking is off—that is, when a procedure or function call is compiled in the {**$I-**} state—an I/O error does not cause the program to halt. To check the result of an I/O operation, you must instead call the standard function *IOResult*.

I/O functions

Function	Description
Eof	Returns the end-of-file status of a file.
FilePos	Returns the current file position of a file. Not used for text files.
FileSize	Returns the current size of a file. Not used for text files.
IOResult	Returns an integer value that is the status of the last I/O function performed.

I/O procedures

Procedure	Description
Assign	Assigns the name of an external file to a file variable.
ChDir	Changes the current directory.
Close	Closes an open file.
Erase	Erases an external file.
GetDir	Returns the current directory of a specified drive.
MkDir	Creates a subdirectory.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.

RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a file to a specified component. Not used with text files.
Truncate	Truncates the file size at the current file position. Not used with text files.

Text files

In Turbo Pascal the type Text is distinct from the type ***file of*** *Char.*

This section summarizes input and output using file variables of the standard type Text.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character).

For text files, there are special forms of *Read* and *Write* that allow you to read and write values that are not of type Char. Such values are automatically translated to and from their character representation. For example, *Read(F, I)*, where *I* is a type Integer variable, will read a sequence of digits, interpret that sequence as a decimal integer, and store it in *I*.

Turbo Pascal defines two standard text-file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input file (typically the keyboard), and the standard file variable *Output* is a write-only file associated with the operating system's standard output file (typically the display).

Since Windows doesn't directly support text-oriented input and output, the *Input* and *Output* files are by default unassigned in a Windows application, and any attempt to read or write to them will produce an I/O error. However, if an application uses the *WinCrt* unit, *Input* and *Output* will refer to a scrollable text window. *WinCrt* contains the complete control logic required to emulate a text screen in the Windows environment, and no "Windows-specific" programming is required in an application that uses *WinCrt*.

Some of the standard procedures and functions listed in this section do not need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* are assumed by default, depending on whether the procedure or function is input- or output-oriented. For instance, *Read(X)*

corresponds to *Read(Input, X)* and *Write(X)* corresponds to *Write(Output, X)*.

If you do specify a file when calling one of the procedures or functions in this section, the file must have been associated with an external file using *Assign*, and opened using *Reset*, *Rewrite*, or *Append*. An error message is generated if you pass a file that was opened with *Reset* to an output-oriented procedure or function. Likewise, it's an error to pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

Procedures

Procedure	Description
Append	Opens an existing file for appending.
Flush	Flushes the buffer of an output file.
Read	Reads one or more values from a text file into one or more variables.
Readln	Does what a *Read* does and then skips to the beginning of the next line in the file.
SetTextBuf	Assigns an I/O buffer to a text file.
Write	Writes one or more values to a text file.
Writeln	Does the same as a *Write*, and then writes an end-of-line marker to the file.

Functions

Function	Description
Eoln	Returns the end-of-line status of a file.
SeekEof	Returns the end-of-file status of a file.
SeekEoln	Returns the end-of-line status of a file.

Untyped files

Untyped files are low-level I/O channels primarily used for direct access to any disk file regardless of type and structuring. An untyped file is declared with the word **file** and nothing more; for example,

```
var
  DataFile: file;
```

For untyped files, the *Reset* and *Rewrite* procedures allow an extra parameter to specify the record size used in data transfers.

For historical reasons, the default record size is 128 bytes. The preferred record size is 1, because that is the only value that correctly reflects the exact size of any file (no partial records are possible when the record size is 1).

Except for *Read* and *Write*, all typed file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

Procedures

Procedure	Description
BlockRead	Reads one or more records into a variable.
BlockWrite	Writes one or more records from a variable.

The FileMode variable

The *FileMode* variable defined by the *System* unit determines the access code to pass to DOS when typed and untyped files (not text files) are opened using the *Reset* procedure.

New files created using Rewrite are always opened in Read/Write mode, corresponding to FileMode = 2.

The default *FileMode* is 2, which allows both reading and writing. Assigning another value to *FileMode* causes all subsequent *Resets* to use that mode.

The range of valid *FileMode* values depends on the version of DOS in use. However, for all versions, the following modes are defined:

0: Read only
1: Write only
2: Read/Write

DOS version 3.x and higher defines additional modes, which are primarily concerned with file-sharing on networks. (For more details,see your DOS programmer's reference manual.)

Devices in Turbo Pascal

Turbo Pascal and the DOS operating system regard external hardware, such as the keyboard, the display, and the printer, as *devices*. From the programmer's point of view, a device is treated as a file, and is operated on through the same standard procedures and functions as files.

Turbo Pascal supports two kinds of devices: DOS devices and text file devices.

DOS devices

DOS devices are implemented through reserved file names that have a special meaning attached to them. DOS devices are completely transparent—in fact, Turbo Pascal is not even aware when a file variable refers to a device instead of a disk file. For example, the program

```
var
  Lst: Text;
begin
  Assign(Lst, 'LPT1');
  Rewrite(Lst);
  Writeln(Lst, 'Hello World...');
  Close(Lst);
end.
```

writes the string `Hello World...` on the printer, even though the syntax for doing so is exactly the same as for a disk file.

The devices implemented by DOS are used for obtaining or presenting legible input or output. Therefore, DOS devices are normally used only in connection with text files. On rare occasions, untyped files can also be useful for interfacing with DOS devices.

In general, you should avoid using DOS devices under Windows and you should use the device I/O functions provided by the Windows API instead. Although some DOS devices, such as LPT1, may work, others, such as CON, will not function properly.

Text file devices

In addition to the CRT device, you can write your own text file device drivers. Read more about this in the "Text file device drivers" section in Chapter 19.

Text file devices are used to implement devices unsupported by DOS or to provide another set of features similar to those supplied by another DOS device. A good example of a text file device is the CRT window implemented by the *WinCrt* standard unit. It provides a terminal-like text screen in a window and allows you to create "Standard I/O" applications under Windows with a minimum of effort.

Unlike DOS devices, text file devices have no reserved file names; in fact, they have no file names at all. Instead, a file is associated with a text file device through a customized *Assign* procedure. For

instance, the *WinCrt* standard unit implements an *AssignCrt* procedure that associates text files with the CRT window.

Predeclared variables

Besides procedures and functions, the *System* unit provides a number of predeclared variables.

Uninitialized variables

Variable	Type	Description
Input	Text	Input standard file
Output	Text	Output standard file

Initialized variables

Variable	Type	Initial value	Description
CmdLine	PChar	**nil**	Command line pointer
CmdShow	Integer	0	CmdShow parameter for CreateWindow
ErrorAddr	Pointer	**nil**	Run-time error address
ExitCode	Integer	0	Exit code
ExitProc	Pointer	**nil**	Exit procedure
FileMode	Byte	2	File open mode
HeapBlock	Word	8192	Heap block size
HeapError	Pointer	**nil**	Heap error function
HeapLimit	Word	1024	Heap small block limit
HeapList	Word	0	Heap segment list
HInstance	Word	0	Handle of this instance
HPrevInst	Word	0	Handle of previous instance
InOutRes	Integer	0	I/O result buffer
PrefixSeg	Word	0	Program segment prefix
RandSeed	Longint	0	Random seed

HInstance contains the instance handle of the application or library as provided by the Windows environment. In a program, *HPrevInst* contains the handle of the previous instance of the application, or zero if there are no previous instances. In a library, *HPrevInst* is always zero.

In a program, *CmdLine* contains a pointer to a null-terminated string that contains the command-line arguments specified when the application was started. In a library, *CmdLine* is undefined.

In a program, *CmdShow* contains the parameter value that Windows expects to be passed to *ShowWindow* when the

application creates its main window. In a library, *CmdShow* is always zero.

The heap manager uses *HeapList*, *HeapLimit*, *HeapBlock*, and *HeapError* to implement Turbo Pascal's dynamic memory allocation routines. The heap manager is described in full in Chapter 16, "Memory issues."

The *ExitProc*, *ExitCode*, and *ErrorAddr* variables are used to implement exit procedures. This is also described in Chapter 18, "Control issues."

In a program, the *PrefixSeg* variable contains the selector (segment address) of the Program Segment Prefix (PSP) created by DOS and Windows when the application was executed. For a complete description of the PSP, refer to your DOS and Windows manuals. In a library, *PrefixSeg* is always zero.

The built-in I/O routines use *InOutRes* to store the value that the next call to the *IOResult* standard function will return.

RandSeed stores the built-in random number generator's seed. By assigning a specific value to *RandSeed*, the *Random* function can be made to generate a specific sequence of random numbers over and over. This is useful for applications that deal with data encryption, statistics, and simulations.

For further details, see page 149.

The *FileMode* variable allows you to change the access mode in which typed and untyped files are opened (by the *Reset* standard procedure).

For further details, see page 145.

Input and *Output* are the standard I/O files required by every Pascal implementation.

C H A P T E R

12

The WinDos unit

The *WinDos* unit implements a number of operating system and file-handling routines. None of the routines in the *WinDos* unit are defined by standard Pascal, so they have been placed in their own module.

For a complete description of DOS operations, refer to the IBM DOS technical manual.

Constants, types, and variables

Each of the constants, types, and variables defined by the *WinDos* unit are briefly discussed in this section. For more detailed information, see the descriptions of the procedures and functions that depend on these objects in Chapter 24, "The run-time library."

Constants

Flag constants

The following constants test individual flag bits in the Flags register after a call to *Intr* or *MsDos*:

Constants	Value
fCarry	$0001
fParity	$0004
fAuxiliary	$0010
fZero	$0040
fSign	$0080
fOverflow	$0800

For instance, if *R* is a register record, the tests

```
R.Flags and fCarry <> 0
R.Flags and fZero = 0
```

are True respectively if the Carry flag is set and if the Zero flag is clear.

File mode constants

The file-handling procedures use these constants when opening and closing disk files. The mode fields of Turbo Pascal's file variables will contain one of the values specified in the following:

Constant	Value
fmClosed	$D7B0
fmInput	$D7B1
fmOutput	$D7B2
fmInOut	$D7B3

File attribute constants

These constants test, set, and clear file attribute bits in connection with the *GetFAttr*, *SetFAttr*, *FindFirst*, and *FindNext* procedures:

Constant	Value
faReadOnly	$01
faHidden	$02
faSysFile	$04
faVolumeID	$08
faDirectory	$10
faArchive	$20
faAnyFile	$3F

The constants are additive, that is, the statement

```
FindFirst('*.*', faReadOnly + faDirectory, S);
```

will locate all normal files as well as read-only files and subdirectories in the current directory. The *faAnyFile* constant is simply the sum of all attributes.

File name component string lengths

These constants are the maximum file name component string lengths used by the functions *FileSearch* and *FileExpand*.

Constant	Value
fsPathName	*79*
fsDirectory	*67*
fsFileName	*8*
fsExtension	*4*

Return flags for FileSplit

These return flags are used by the function *FileSplit*. The returned value is a combination of the *fcDirectory*, *fcFileName*, and *fcExtension* bit masks, indicating which components were present in the path. If the name or extension contains any wilcard characters (* or ?), the *fcWildcards* flag is set in the return value.

Constant	Value
fcExtension	*$0001*
fcFileName	*$0002*
fcDirectory	*$0004*
fcWildcards	*$0008*

Types

File record types

The record definitions used internally by Turbo Pascal are also declared in the *WinDos* unit. *TFileRec* is used for both typed and untyped files, while *TTextRec* is the internal format of a variable of type text.

```
type
{ Typed and untyped files }
  TFileRec = record
    Handle: Word;
    Mode: Word;
    RecSize: Word;
    Private: array[1..26] of Byte;
    UserData: array[1..16] of Byte;
    Name: array[0..79] of Char;
  end;

{ Textfile record }
  PTextBuf = ^TTextBuf;
  TTextBuf = array[0..127] of Char;
  TTextRec = record
    Handle: Word;
```

```
    Mode: Word;
    BufSize: Word;
    Private: Word;
    BufPos: Word;
    BufEnd: Word;
    BufPtr: ^TTextBuf;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..16] of Byte;
    Name: array[0..79] of Char;
    Buffer: TTextBuf;
  end;
```

The TRegisters type

The *Intr* and *MsDos* procedures use variables of type *TRegisters* to specify the input register contents and examine the output register contents of a software interrupt.

```
type
  TRegisters = record
    case Integer of
    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
    1: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
  end;
```

Notice the use of a variant record to map the 8-bit registers on top of their 16-bit equivalents.

The TDateTime type

Variables of *TDateTime* type are used in connection with the *UnpackTime* and *PackTime* procedures to examine and construct 4-byte, packed date-and-time values for the *GetFTime, SetFTime, FindFirst,* and *FindNext* procedures.

```
type
  TDateTime = record
    Year,Month,Day,Hour,Min,Sec: Word;
  end;
```

Valid ranges are *Year* 1980..2099, *Month* 1..12, *Day* 1..31, *Hour* 0..23, *Min* 0..59, and *Sec* 0..59.

The TSearchRec type

The *FindFirst* and *FindNext* procedures use variables of type *TSearchRec* to scan directories.

```
type
  TSearchRec = record
```

```
    Fill: array[1..21] of Byte;
    Attr: Byte;
    Time: Longint;
    Size: Longint;
    Name: array[0..12] of Char;
  end;
```

The information for each file found by one of these procedures is reported back in a *TSearchRec*. The *Attr* field contains the file's attributes (constructed from file attribute constants), *Time* contains its packed date and time (use *UnpackTime* to unpack), *Size* contains its size in bytes, and *Name* contains its name. The *Fill* field is reserved by DOS and should never be modified.

Variables

The DosError variable

DosError is used by many of the routines in the *WinDos* unit to report errors.

```
var DosError: Integer;
```

The values stored in *DosError* are DOS error codes. A value of 0 indicates no error; other possible error codes include the following:

DOS error code	Meaning
2	File not found
3	Path not found
5	Access denied
6	Invalid handle
8	Not enough memory
10	Invalid environment
11	Invalid format
18	No more files

Date and time procedures

Procedure	Description
GetDate	Returns the current date set in the operating system.
GetFTime	Returns the date and time a file was last written.
GetTime	Returns the current time set in the operating system.
PackTime	Converts a *TDateTime* record into a 4-byte, packed date-and-time character Longinit used by *SetFTime*.
SetDate	Sets the current date in the operating system.
SetFTime	Sets the date and time a file was last written.
SetTime	Sets the current time in the operating system.
UnpackTime	Converts a 4-byte, packed date-and-time character Longint returned by *GetFTime*, *FindFirst*, or *FindNext* into an unpacked *TDateTime* record.

Interrupt support procedures

Procedure	Description
GetIntVec	Returns the address stored in a specified interrupt vector.
Intr	Executes a specified software interrupt with a specified *TRegisters* package.
MsDos	Executes a DOS function call with a specified *TRegisters* package.
SetIntVec	Sets a specified interrupt vector to a specified address.

Disk status functions

Function	Description
DiskFree	Returns the number of free bytes of a specified disk drive.
DiskSize	Returns the total size in bytes of a specified disk drive.

File-handling procedures

Procedure	Description
FindFirst	Searches the specified (or current) directory for the first entry matching the specified file name and set of attributes.
FindNext	Returns the next entry that matches the name and attributes specified in a previous call to *FindFirst*.

GetFAttr	Returns the attributes of a file.
SetFAttr	Sets the attributes of a file.

File-handling functions

Function	Description
FileSplit	Splits a file name into its three component parts (directory, file name, and extension).
FileExpand	Takes a file name and returns a fully qualified file name (drive, directory, and extension).
FileSearch	Searches for a file in a list of directories.

Directory-handling procedures

Function	Description
CreateDir	Creates a new subdirectory.
RemoveDir	Removes a subdirectory.
SetCurDir	Changes the current directory.

Directory-handling procedures

Function	Description
GetCurDir	Returns the current directory of a specified drive.

Environment-handling functions

Function	Description
GetArgCount	Returns the number of parameters passed to the program on the command line.
GetArgStr	Returns a specified environment string.
GetEnvVar	Returns a pointer to the value of a specified environment variable.

Miscellaneous procedures

Procedure	Description
GetCBreak	Returns the state of *Ctrl-Break* checking in DOS.
GetVerify	Returns the state of the verify flag in DOS.
SetCBreak	Sets the state of *Ctrl-Break* checking in DOS.
SetVerify	Sets the state of the verify flag in DOS.

Miscellaneous functions

Function	Description
DosVersion	Returns the DOS version number.

C H A P T E R

13

The Strings unit

Turbo Pascal for Windows introduces support for a new class of character strings called *null-terminated strings*. With Turbo Pascal's extended syntax and the *Strings* unit, your programs can use null-terminated strings, the format required by the Windows Application Programming Interface (API).

What is a null-terminated string?

The compiler stores a "traditional" Turbo Pascal **string** type as a length byte followed by a sequence of characters. The maximum length of a Pascal string is 255 characters, and a Pascal string occupies from 1 to 256 bytes of memory.

A null-terminated string has no length byte; instead, it consists of a sequence of non-null characters followed by a NULL (#0) character. There is no inherent restriction on the length of a null-terminated string, but the 16-bit architecture of DOS and Windows does impose an upper limit of 65,535 characters.

You can use the *StrPCopy* to copy a Pascal string to a null-terminated string, and you can use *StrPas* to convert a null-terminated string to a Pascal string.

Using null-terminated strings

Null-terminated strings are stored as arrays of characters with a zero-based integer index type; that is, an array of the form

```
array[0..X] of Char
```

where *X* is a positive nonzero integer. Such arrays are referred to as *zero-based character arrays*. Here are some examples of declarations of zero-based character arrays that can be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..79] of Char;
  TMemoText = array[0..1023] of Char;
```

The biggest difference between using Pascal strings and null-terminated strings is the extensive use of *pointers* in the manipulation of null-terminated strings. Turbo Pascal for Windows performs operations on these pointers with a set of *extended syntax* rules. Also, Turbo Pascal has a new predefined type, PChar, to represent a pointer to a null-terminated string. The *System* unit declares PChar as

```
type PChar = ^Char;
```

The **$X** compiler directive controls the extended syntax rules. When **$X** is on ({**$X+**}, the default state) Turbo Pascal's extended syntax is enabled. The extended syntax rules are outlined in the following sections.

Character pointers and string literals

When extended syntax is enabled, a string literal is *assignment compatible* with the PChar type. This means that a string literal can be assigned to a variable of type PChar. For example,

```
var
  P: PChar;
  .
  .
begin
  P := 'Hello world...';
end;
```

The effect of such an assignment is that the pointer points to an area of memory that contains a zero-terminated copy of the string

literal. The compiler stores the string literal in the data segment, much like a "hidden" typed constant declaration:

```
const
  TempString: array[0..14] of Char = 'Hello world...'#0;
var
  P: PChar;
  .
  .
begin
  P := @TempString;
end;
```

You can use string literals as actual parameters in procedure and function calls when the corresponding formal parameter is of type *Char. For example, given a procedure with the declaration*

```
procedure PrintStr(Str: PChar);
```

the following procedure calls are valid:

```
PrintStr('This is a test');
PrintStr(#10#13);
```

Just as it does with an assignment, the compiler generates a zero-terminated copy of the string literal in the data segment and passes a pointer to that memory area in the *Str* parameter of the *PrintStr* procedure.

Finally, a typed constant of type PChar can be initialized with a string constant. This feature extends to structured types as well, such as arrays of PChar and records and objects with PChar fields.

```
const
  Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Character pointers and character arrays

When you enable extended syntax with **$X**, a zero-based character array is *compatible* with the PChar type. This means that whenever a PChar is expected, a zero-based character array can be used instead. When a character array is used in place of a PChar value, the compiler converts the character array to a pointer *constant* whose value corresponds to the address of the first element of the array.

```
var
  A: array[0..63] of Char;
  P: PChar;
  .
  .
begin
  P := A;
  PrintStr(A);
  PrintStr(P);
end;
```

Because of this assignment statement, *P* now points to the first element of *A*, so *PrintStr* is called twice with the same value.

You can initialize a typed constant of a zero-based character array type with a string literal that is shorter than the declared length of the array. The remaining characters are set to NULL (#0) and the array effectively contains a null-terminated string.

```
type
  TFileName = array[0..79] of Char;
const
  FileNameBuf: TFileName = 'TEST.PAS';
  FileNamePtr: PChar = FileNameBuf;
```

Character pointer indexing

Just as a zero-based character array is compatible with a character pointer, so can a character pointer be indexed as if it were a zero-based character array.

```
var
  A: array[0..63] of Char;
  P: PChar;
  Ch: Char;
  .
  .
begin
  P := A;
  Ch := A[5];
  Ch := P[5];
end;
```

Both of the last two assignments assign *Ch* the value contained in the sixth character element of *A*.

When you index a character pointer, the index specifies an unsigned *offset* to add to the pointer before it is dereferenced. Thus, *P[0]* is equivalent to *P^* and specifies the character pointed

to by *P*. *P[1]* specifies the character right after the one pointed to by *P*, *P[2]* specifies the next character, and so on. For purposes of indexing, a PChar behaves as if it were declared as

```
type
  TCharArray = array[0..65535] of Char;
  PChar = ^TCharArray;
```

The compiler performs no range checks when indexing a character pointer because it has no type information available to determine the maximum length of the null-terminated string pointed to by the character pointer. Your program must perform any such range checking.

The *StrUpper* function shown here illustrates the use of character pointer indexing to convert a null-terminated string to uppercase.

```
function StrUpper(Str: PChar): PChar;
var
  I: Word;
begin
  I := 0;
  while Str[I] <> #0 do
  begin
     Str[I] := UpCase(Str[I]);
     Inc(I);
  end;
  StrUpper := Str;
end;
```

Notice that *StrUpper* is a function, not a procedure, and that it always returns the value that it was passed as a parameter. Since the extended syntax allows the result of a function call to be ignored, *StrUpper* can be treated as if it were a procedure:

```
StrUpper(A);
PrintStr(A);
```

However, as *StrUpper* always returns the value it was passed, the preceding statements can be combined into one:

```
PrintStr(StrUpper(A));
```

Nesting calls to null-terminated string-handling functions can be very convenient when you want to indicate a certain interrelationship between a set of sequential string manipulations.

Character pointer operations

Turbo Pascal's extended syntax permits new operations on character pointers. The plus (+) and minus (-) operators can be used to increment and decrement the offset part of a pointer value, and the minus operator can be used to calculate the distance (difference) between the offset parts of two character pointers. Assuming that *P* and *Q* are values of type PChar and *I* is a value of type *Word*, the following constructs are allowed:

P + *I*	Add *I* to the offset part of *P*
I + *P*	Add *I* to the offset part of *P*
P – *I*	Subtract *I* from the offset part of *P*
P – *Q*	Subtract offset part of *Q* from offset part of *P*

The operations *P* + *I* and *I* + *P* adds *I* to the address given by *P*, producing a pointer that points *I* characters after *P*. The operation *P* – *I* subtracts *I* from the address given by *P*, producing a pointer that points *I* characters before *P*.

The operation *P* – *Q* computes the distance between *Q* (the lower address) and *P* (the higher address), producing a value of type *Word* that gives the number of characters between *Q* and *P*. This operation assumes that *P* and *Q* *point within the same character array*. If the two character pointers point into different character arrays, the result is undefined.

Standard Turbo Pascal syntax allows pointers to be compared only to see if they are equal or not. The extended syntax also allows the >, <, >=, and <= operators to be applied to PChar values. Note, however, that these relational tests assume the two pointers being compared *point within the same character array*, and for that reason, the operators only compare the offset parts of the two pointer values. If the two character pointers point into different character arrays, the result is undefined.

```
var
  A, B: array[0..79] of Char;
  P, Q: PChar;
begin
  P := A;                  { P points to A[0] }
  Q := A + 5;              { Q points to A[5] }
  if P < Q then ...;       { Valid test, result is True }
  Q := B;                  { Q points to B[0] }
  if P < Q then ...;       { Result is undefined }
end;
```

Null-terminated strings and standard procedures

Turbo Pascal's extended syntax allows the *Read*, *Readln*, and *Val* standard procedures to be applied to zero-based character arrays, and allows the *Write*, *Writeln*, *Val*, *Assign*, and *Rename* standard procedures to be applied to both zero-based character arrays and character pointers. For further details, refer to the descriptions of these standard procedures in Chapter 24.

Using the Strings unit

See Chapter 24, "The run-time library." for more detailed descriptions of each function.

Turbo Pascal for Windows has no built-in routines specifically for null-terminated string handling. Instead you'll find all such functions in the *Strings* unit. Here is a brief description of each function.

Table 13.1
Functions in the Strings unit

Function	Description
StrLen	Returns the length of a string.
StrEnd	Returns a pointer to the end of a string, that is, a pointer to the null character that terminates a string.
StrMove	Moves a block of characters from a source string to a destination string, and returns a pointer to the destination string. The two blocks may overlap.
StrCopy	Copies a source string to a destination string and returns a pointer to the destination string.
StrECopy	Copies a source string to a destination string and returns a pointer to the end of the destination string.
StrLCopy	Copies up to a given number of characters from a source string to a destination string and returns a pointer to the destination string.
StrPCopy	Copies a Pascal string to a null-terminated string and returns a pointer to the null-terminated string.
StrCat	Appends a source string to the end of a destination string and returns a pointer to the destination string.
StrLCat	Appends a source string to the end of a destination string, ensuring that the length of the resulting string does not exceed a given maximum, and returns a pointer to the destination string.

Table 13.1: Functions in the Strings unit (continued)

StrComp	Compares two strings, *S1* and *S2*, and returns a value less than zero if *S1* < *S2*, zero if *S1* = *S2*, or greater than zero if *S1* > *S2*.
StrIComp	Compares two strings without case sensitivity.
StrLComp	Compares two strings for a given maximum length.
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrPas	Converts a null-terminated string to a Pascal string.
StrPos	Returns a pointer to the first occurrence of a given substring within a string, or **nil** if the substring does not occur within the string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string, or **nil** if the character does not occur within the string.
StrScan	Returns a pointer to the first occurrence of a given character within a string, or **nil** if the character does not occur within the string.
StrUpper	Converts a string to uppercase and returns a pointer to the string.
StrLower	Converts a string to lowercase and returns a pointer to the string.
StrNew	Allocates a string on the heap.
StrDispose	Disposes of a previously allocated string.

Here is a code example that shows you how we used some of the string-handling functions when we wrote the *FileSplit* function in the *WinDos* unit.

```
{ Maximum file name component string lengths }

const
  fsPathName  = 79;
  fsDirectory = 67;
  fsFileName  = 8;
  fsExtension = 4;

{ FileSplit return flags }

const
  fcExtension = $0001;
  fcFileName  = $0002;
  fcDirectory = $0004;
  fcWildcards = $0008;

{ FileSplit splits the file name specified by Path into its     }
```

```
{ three components. Dir is set to the drive and directory path  }
{ with any leading and trailing backslashes, Name is set to the }
{ file name, and Ext is set to the extension with a preceding   }
{ period. If a component string parameter is NIL, the           }
{ corresponding part of the path is not stored. If the path     }
{ does not contain a given component, the returned component    }
{ string is empty. The maximum lengths of the strings returned  }
{ in Dir, Name, and Ext are defined by the fsDirectory,         }
{ fsFileName, and fsExtension constants. The returned value is  }
{ a combination of the fcDirectory, fcFileName, and fcExtension }
{ bit masks, indicating which components were present in the    }
{ path. If the name or extension contains any wildcard          }
{ characters (* or ?), the fcWildcards flag is set in the       }
{ returned value.                                               }

function FileSplit(Path, Dir, Name, Ext: PChar): Word;
var
  DirLen, NameLen, Flags: Word;
  NamePtr, ExtPtr: PChar;
begin
  NamePtr := StrRScan(Path, '\');
  if NamePtr = nil then NamePtr := StrRScan(Path, ':');
  if NamePtr = nil then NamePtr := Path else Inc(NamePtr);
  ExtPtr := StrScan(NamePtr, '.');
  if ExtPtr = nil then ExtPtr := StrEnd(NamePtr);
  DirLen := NamePtr - Path;
  if DirLen > fsDirectory then DirLen := fsDirectory;
  NameLen := ExtPtr - NamePtr;
  if NameLen > fsFilename then NameLen := fsFilename;
  Flags := 0;
  if (StrScan(NamePtr, '?') <> nil) or
    (StrScan(NamePtr, '*') <> nil) then
    Flags := fcWildcards;
  if DirLen <> 0 then Flags := Flags or fcDirectory;
  if NameLen <> 0 then Flags := Flags or fcFilename;
  if ExtPtr[0] <> #0 then Flags := Flags or fcExtension;
  if Dir <> nil then StrLCopy(Dir, Path, DirLen);
  if Name <> nil then StrLCopy(Name, NamePtr, NameLen);
  if Ext <> nil then StrLCopy(Ext, ExtPtr, fsExtension);
  FileSplit := Flags;
end;
```

C H A P T E R

14

The WinCrt unit

The *WinCrt* unit implements a terminal-like text screen in a window. With *WinCrt*, you can easily create a program that uses the *Read, Readln, Write,* and *Writeln* standard procedures to perform input and output operations just as you would in a traditional text-mode application. *WinCrt* contains the complete control logic required to emulate a text screen in the Windows environment. You don't need to write "Windows-specific" code if your program uses *WinCrt*.

Using the WinCrt unit

To use the *WinCrt* unit, simply include it in your program's **uses** clause, just as you would any other unit:

```
uses WinCrt;
```

By default, the *Input* and *Output* standard text files defined in the *System* unit are unassigned, and any *Read, Readln, Write,* or *Writeln* procedure call without a file variable causes an I/O error to occur. However, when a program uses the *WinCrt* unit, the initialization code of the unit assigns the *Input* and *Output* standard text files to refer to a window that emulates a text screen. It's as if the following statements are executed at the beginning of the program:

```
AssignCrt(Input); Reset(Input);
AssignCrt(Output); Rewrite(Output);
```

When the first *Read*, *Readln*, *Write*, or *Writeln* call executes in the program, a CRT window opens on the Windows desktop. The default title of a CRT window is the full path of the program's .EXE file. When the program finishes (when control reaches the final **end** reserved word) the title of the CRT window is changed to "(Inactive *nnnnn*)", where *nnnnn* is the title of the window in its active state.

Note that even though the program has finished, the window stays up so that the user can examine the program's output. Just like any other Windows application, the program doesn't completely terminate until the user closes the window.

The *InitWinCrt* and *DoneWinCrt* routines give you greater control over the CRT window's life cycle. A call to *InitWinCrt* creates the CRT window immediately rather than waiting for the first call to *Read*, *Readln*, *Write*, or *Writeln*. Likewise, calling *DoneWinCrt* destroys the CRT window immediately instead of when the user closes it.

The CRT window is a scrollable "panning" window on a virtual text screen. The default dimensions of the virtual text screen are 80 columns by 25 lines, but the actual size of the CRT window may be less. If the size is less, the user can use the window's scroll bars or the cursor keys to move this panning window over the larger text screen. This is particularly useful for "scrolling back" to examine previously written text. By default, the panning window tracks the text screen cursor. In other words, the panning window automatically scrolls to ensure that the cursor is always visible. You can disable the autotracking feature by setting the *AutoTracking* variable to *False*.

The dimensions of the virtual text screen are determined by the *ScreenSize* variable. You can change the virtual screen dimensions by assigning new dimensions to *ScreenSize* before your program creates the CRT window. When the window is created, a screen buffer is allocated in dynamic memory. The size of this buffer is *ScreenSize.X* multiplied by *ScreenSize.Y*, and it cannot be larger than 65,520 bytes. It is up to you to ensure that the values you assign to *ScreenSize.X* and *ScreenSize.Y* do not overflow this limit. If, for example, you assign 64 to *ScreenSize.X*, the largest allowable value for *ScreenSize.Y* is 1,023.

At any time while running a program that uses the *WinCrt* unit, the user can terminate the application by choosing the Close command on the CRT window's Control menu, double-clicking

the Control menu box, or pressing *Alt+F4*. Likewise, the user can press *Ctrl+C* or *Ctrl+Break* at any time to halt the application and force the window into its inactive state. You can disable these features by setting the *CheckBreak* variable to *False* at the beginning of the program.

Special characters

When writing to *Output* or a file that has been assigned to the CRT window, the following control characters have special meanings:

Char	Name	Description
#7	BELL	Emits a beep from the internal speaker.
#8	BS	Moves the cursor left one column and erases the character at that position. If the cursor is already at the left edge of the screen, nothing happens.
#10	LF	Moves the cursor down one line. If the cursor is already at the bottom of the virtual screen, the screen is scrolled up one line.
#13	CR	Returns the cursor to the left edge of the screen.

Line input

When your program reads from *Input* or a file that has been assigned to the CRT window, text is input one line at a time. The line is stored in the text file's internal buffer, and when variables are read, this buffer is used as the input source. When the buffer empties, a new line is stored in the buffer.

When entering lines in the CRT window, the user can use the *Backspace* key to delete the last character entered. Pressing *Enter* terminates the input line and stores an end-of-line marker (CR/LF) in the buffer. In addition, if the *CheckEOF* variable is set to *True*, a *Ctrl+Z* also terminates the input line and generates an end-of-file marker. *CheckEOF* is *False* by default.

To test keyboard status and input single characters under program control, use the *KeyPressed* and *ReadKey* functions.

Variables

The *WinCrt* unit declares several typed constants and one variable. The typed constants have initial values. You can regard them as variables, however, because you can change the initial values of most of them to suit your needs.

Variable	Type
WindowOrg	TPoint
WindowSize	TPoint
ScreenSize	TPoint
Cursor	TPoint
Origin	TPoint
InactiveTitle	PChar
AutoTracking	Boolean
CheckEOF	Boolean
CheckBreak	Boolean
WindowTitle	array [0..79] of Char

WindowOrg

Determines the initial location of the CRT window.

```
const WindowOrg: TPoint = (X: cw_UseDefault; Y: cw_UseDefault);
```

The default location allows Windows to select a suitable location for the CRT window. You can change the initial location by assigning new values to the *X* and *Y* coordinates before the CRT window is created.

WindowSize

Determines the initial size of the CRT window.

```
const WindowSize: TPoint = (X: cw_UseDefault; Y: cw_UseDefault);
```

The default size allows Windows to select a suitable size for the CRT window. You can change the initial size by assigning new values to the *X* and *Y* coordinates before the CRT window is created.

ScreenSize

Determines the width and height in characters of the virtual screen within the CRT window.

```
const ScreenSize: TPoint = (X: 80; Y: 25);
```

The default screen size is 80 columns by 25 lines. You can change the size of the virtual screen by assigning other values to the *X* and *Y* coordinates of *ScreenSize* before the CRT window is created.

The value given by *ScreenSize.X* multiplied by *ScreenSize.Y* must not exceed 65,520.

Cursor

Contains the current position of the cursor within the virtual screen.

```
const Cursor: TPoint = (X: 0; Y: 0);
```

The upper left corner corresponds to (0, 0). *Cursor* is a read-only variable; do not assign values to it.

Origin

Contains the virtual screen coordinates of the character cell displayed in the upper-left corner of the CRT window.

```
const Origin: TPoint = (X: 0; Y: 0);
```

Origin is a read-only variable; do not assign values to it.

InactiveTitle

Points to a null-terminated string to use when constructing the title of an inactive CRT window.

```
const InactiveTitle: PChar = '(Inactive %s)';
```

The string is used as the format-control parameter of a call to the Windows *WVSPrintF* function. The *%s* specifier, if present, indicates where to insert the existing window title.

AutoTracking

Enables and disables the automatic scrolling of the window to keep the cursor visible.

```
const AutoTracking: Boolean = True;
```

When *AutoTracking* is *True*, the CRT window automatically scrolls to ensure that the cursor is visible after each *Write* and *Writeln*. If *AutoTracking* is *False*, the CRT window will not scroll automatically and text written to the window may not be visible to the user.

CheckEOF

Enables and disables the end-of-file character.

```
const CheckEOF: Boolean = False;
```

When *CheckEOF* is *True*, an end-of-file marker is generated when the user presses *Ctrl+Z* while reading from a file assigned to the CRT window. When *CheckEOF* is *False*, pressing *Ctrl+Z* has no effect.

CheckBreak Enables and disables user termination of an application.

```
const CheckBreak: Boolean = True;
```

When *CheckBreak* is *True*, the user can terminate the application at any time by choosing the Close command on the CRT window's Control menu, double-clicking the window's Control-menu box, or by pressing *Alt+F4*. Likewise, the user can press *Ctrl+C* or *Ctrl+Break* at any time to halt the application and force the CRT window into its inactive state. All of these features are disabled when *CheckBreak* is *False*.

WindowTitle Determines the title of the CRT window.

```
var WindowTitle: array[0..79] of Char;
```

The default value is the full path of the program's .EXE file. You can change the title by storing a new string in *WindowTitle* before the CRT window is created. Here is an example:

```
StrCopy(WindowTitle, 'Hello World');
```

Procedures and Functions

The following tables list the procedures and functions used in the *WinCrt* unit.

Procedure	Description
AssignCrt	Associates a text file with the CRT window.
ClrEol	Clears all the characters from the cursor position to the end of the line.
ClrScr	Clears the screen and returns cursor to upper left-hand corner.
CursorTo	Moves the cursor to the given coordinates within the virtual screen.
DoneWinCrt	Destroys the CRT window.
GotoXY	Moves the cursor to the given coordinates within the virtual screen.
InitWinCrt	Creates the CRT window.
ScrollTo	Scrolls the CRT window to show a screen location.
TrackCursor	Scrolls the CRT window to keep cursor visible.

WriteBuf	Writes a block of characters to the CRT window.
WriteChar	Writes a single character to the CRT window.

Function	Description
KeyPressed	Returns *True* if a key has been pressed on the keyboard.
ReadBuf	Inputs a line from the CRT window.
ReadKey	Reads a character from the keyboard.
WhereX	Returns the X coordinate of the current cursor location.
WhereY	Returns the Y coordinate of the current cursor location.

The following text describes the procedures and functions in more depth.

InitWinCrt

Creates the CRT window if it hasn't already been created.

```
procedure InitWinCrt;
```

A *Read, Readln, Write,* or *Writeln* with a file that has been assigned to the CRT automatically calls *InitWinCrt* to ensure that the CRT window exists. *InitWinCrt* uses the *WindowOrg, WindowSize,* and *ScreenSize* constants, and the *WindowTitle* variable to determine the characteristics of the CRT window.

DoneWinCrt

Destroys the CRT window if it hasn't already been destroyed.

```
procedure DoneWinCrt;
```

Calling *DoneWinCrt* just before the program ends prevents the CRT window from entering the inactive state. The user, therefore, is not required to close the window manually.

WriteBuf

Writes a block of characters to the CRT window.

```
procedure WriteBuf(Buffer: PChar; Count: Word);
```

Buffer points to the first character in the block, and *Count* contains the number of characters to write. If *AutoTracking* is *True,* the CRT window scrolls if necessary to ensure that the cursor is visible after writing the block of characters.

WriteChar Writes a single character to the CRT window.

```
procedure WriteChar(Ch: Char);
```

KeyPressed Returns *True* if a key has been pressed on the keyboard, and *False* otherwise.

```
function KeyPressed: Boolean;
```

The key can be read using the *ReadKey* function.

ReadKey Reads a character from the keyboard.

```
function ReadKey: Char;
```

The character is *not* echoed to the screen.

The *ReadKey* function only supports standard ASCII key codes. Extended key codes, such as function and cursor key codes, are not supported by *ReadKey*.

ReadBuf Inputs a line from the CRT window.

```
function ReadBuf(Buffer: PChar; Count: Word): Word;
```

Buffer points to a line buffer that has room for up to *Count* characters. Up to *Count* – 2 characters can be input, and an end-of-line marker (a #13 followed by a #10) is automatically appended to the line when the user presses *Enter*. If *CheckEOF* is *True*, the user can also terminate the input line by pressing *Ctrl+Z*, and the line will in that case have an end-of-file marker (a #26) appended to it. The return value is the number of characters read, including the end-of-line or end-of-file marker.

GotoXY Moves the cursor to the given coordinates within the virtual screen.

```
procedure GotoXY(X, Y: Integer);
```

The upper left-hand corner corresponds to (1, 1). The *Cursor* variable is set to ($X - 1$, $Y - 1$), since it stores the cursor position relative to (0, 0) instead of relative to (1, 1).

The *GotoXY*, *WhereX*, and *WhereY* routines are intended primarily for compatibility with the *Crt* unit provided by Turbo Pascal for DOS. The 1-based coordinates of these routines are inconsistent with the rest of the routines and variables in the *WinCrt* unit, and

we suggest that you instead use the *CursorTo* routine and the *Cursor* variable.

WhereX Returns the *X* coordinate of the current cursor location.

```
function WhereX: Integer;
```

The returned value is 1-based, and corresponds to *Cursor.X* + 1.

WhereY Returns the *Y* coordinate of the current cursor location.

```
function WhereY: Integer;
```

The returned value is 1-based, and corresponds to *Cursor.Y* + 1.

ClrScr Clears the screen and returns the cursor to the upper left-hand corner.

```
procedure ClrScr;
```

ClrEol Clears all characters from the cursor position to the end of the line without moving the cursor.

```
procedure ClrEol;
```

CursorTo Moves the cursor to the given coordinates within the virtual screen.

```
procedure CursorTo(X, Y: Integer);
```

The upper left-hand corner corresponds to (0, 0). The *Cursor* variable is set to (*X*, *Y*).

ScrollTo Scrolls the CRT window to show the virtual screen location given by (*X*,*Y*) in the upper left-hand corner.

```
procedure ScrollTo(X, Y: Integer);
```

(0, 0) corresponds to the upper left-hand corner of the virtual screen. The *Origin* variable is set to (*X*, *Y*).

TrackCursor Scrolls the CRT window if necessary to ensure that the cursor is visible.

```
procedure TrackCursor;
```

AssignCrt Associates a text file with the CRT window.

```
procedure AssignCrt(var F: Text);
```

Subsequent *Write* and *Writeln* operations on the file write to the CRT window, and *Read* and *Readln* operations read from the CRT window.

C H A P T E R

15

Using the 80x87

There are two kinds of numbers you can work with in Turbo Pascal: integers (Shortint, Integer, Longint, Byte, Word) and reals (Real, Single, Double, Extended, Comp). Reals are also known as floating-point numbers. The 80x86 family of processors is designed to handle integer values easily, but handling reals is considerably more difficult. To improve floating-point performance, the 80x86 family of processors has a corresponding family of math coprocessors, the 80x87s.

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly, so if you use floating point a lot, you'll probably want a coprocessor.

Turbo Pascal provides optimal floating-point performance whether or not you have an 80x87.

- For programs running on any PC, with or without an 80x87, Turbo Pascal provides the Real type and an associated library of software routines that handle floating-point operations. The Real type occupies 6 bytes of memory, providing a range of 2.9×10^{-39} to 1.7×10^{38} with 11 to 12 significant digits. The software floating-point library is optimized for speed and size, trading in some of the fancier features provided by the 80x87 processor.
- If you need the added precision and flexibility of the 80x87, you can instruct Turbo Pascal to produce code that uses the 80x87 chip. This gives you access to four additional real types (Single, Double, Extended, and Comp), and an Extended floating-point

range of 3.4×10^{-4951} to 1.1×10^{4932} with 19 to 20 significant digits.

You switch between the two different models of floating-point code generation using the **$N** compiler directive or the 80x87 Code check box in the Options | Compiler dialog box. The default state is {**$N-**}, and in this state, the compiler uses the 6-byte floating-point library, allowing you to operate only on variables of type Real. In the {**$N+**} state, the compiler generates code for the 80x87, giving you increased precision and access to the four additional real types.

When you run an application compiled in the {**$N+**} state, be sure that Windows can find the Windows 8087 Emulation Library, WIN87EM.DLL on your system. The WIN87EM.DLL library provides the necessary interface between the 80x87 processor, Windows, and your application. If an 80x87 processor is not present in your system, WIN87EM.DLL will *emulate* it in software. Emulation is substantially slower than the real 80x87 processor, but it does guarantee that an application using the 80x87 can be run on any machine.

Even though you do have an 80x87 processor in your system, the WIN87EM.DLL emulator library must still be present when running programs compiled in the {**$N+**} state.

Important! When you're compiling in 80x87 Code mode, {**$N+**}, the return values of the floating-point routines in the *System* unit (*Sqrt*, *Pi*, *Sin*, and so on) are of type Extended instead of Real:

```
{$N+}
begin
  Writeln(Pi);                          { 3.14159265358979E+0000 }
end.

{$N-}
begin
  Writeln(Pi)                                 { 3.1415926536E+00 }
end.
```

The remainder of this chapter discusses special issues concerning Turbo Pascal programs that use the 80x87 coprocessor.

The 80x87 data types

For programs that use the 80x87, Turbo Pascal provides four floating-point types in addition to the type Real.

- The Single type is the smallest format you can use with floating-point numbers. It occupies 4 bytes of memory, providing a range of 1.5×10^{-45} to 3.4×10^{38} with 7 to 8 significant digits.
- The Double type occupies 8 bytes of memory, providing a range of 5.0×10^{-324} to 1.7×10^{308} with 15 to 16 significant digits.
- The Extended type is the largest floating-point type supported by the 80x87. It occupies 10 bytes of memory, providing a range of 3.4×10^{-4932} to 1.1×10^{4932} with 19 to 20 significant digits. Any arithmetic involving real-type values is performed with the range and precision of the Extended type.
- The Comp type stores integral values in 8 bytes, providing a range of $-2^{63}+1$ to $2^{63}-1$, which is approximately -9.2×10^{18} to 9.2×10^{18}. Comp may be compared to a double-precision Longint, but it is considered a real type because all arithmetic done with Comp uses the 80x87 coprocessor. Comp is well suited for representing monetary values as integral values of cents or mils (thousandths) in business applications.

Whether or not you have an 80x87, the 6-byte Real type is always available, so you need not modify your source code when switching to the 80x87, and you can still read data files generated by programs that use software floating point.

Note, however, that 80x87 floating-point calculations on variables of type Real are slightly slower than on other types. This is because the 80x87 cannot directly process the Real format—instead, calls must be made to library routines to convert Real values to Extended before operating on them. If you are concerned with optimum speed and never need to run on a system without an 80x87, you may want to use the Single, Double, Extended, and Comp types exclusively.

Extended range arithmetic

The Extended type is the basis of all floating-point computations with the 80x87. Turbo Pascal uses the Extended format to store all

non-integer numeric constants and evaluates all non-integer numeric expressions using extended precision. The entire right side of the following assignment, for instance, will be computed in extended before being converted to the type on the left side:

```
{$N+}
var
  X ,A ,B ,C: Real;
begin
  X := (B + Sqrt(B * B - A * C)) / A;
end;
```

With no special effort by the programmer, Turbo Pascal performs computations using the precision and range of the Extended type. The added precision means smaller round-off errors, and the additional range means overflow and underflow are less common.

You can go beyond Turbo Pascal's automatic Extended capabilities. For example, you can declare variables used for intermediate results to be of type Extended. The following example computes a sum of products:

```
var
  Sum: Single;
  X, Y: array[1..100] of Single;
  I: Integer;
  T: Extended;                          { For intermediate results }
begin
  T := 0.0;
  for I := 1 to 100 do
    T := T + X[I] * Y[I];
  Sum := T;
end;
```

Had *T* been declared Single, the assignment to *T* would have caused a round-off error at the limit of single precision at each loop entry. But because *T* is Extended, all round-off errors are at the limit of extended precision, except for the one resulting from the assignment of *T* to *Sum*. Fewer round-off errors mean more accurate results.

You can also declare formal value parameters and function results to be of type Extended. This avoids unnecessary conversions between numeric types, which can result in loss of accuracy. For example,

```
function Area(Radius: Extended): Extended;
begin
  Area := Pi * Radius * Radius;
end;
```

Comparing reals

Because real-type values are approximations, the results of comparing values of different real types are not always as expected. For example, if *X* is a variable of type Single and *Y* is a variable of type Double, then the following statements will output False:

```
X := 1 / 3;
Y := 1 / 3;
Writeln(X = Y);
```

The reason is that *X* is accurate only to 7 to 8 digits, where *Y* is accurate to 15 to 16 digits, and when both are converted to Extended, they will differ after 7 to 8 digits. Similarly, the statements

```
X := 1 / 3;
Writeln(X = 1 / 3);
```

will output False, since the result of 1/3 in the *Writeln* statement is calculated with 20 significant digits.

The 80x87 evaluation stack

The 80x87 coprocessor has an internal evaluation stack that can be up to eight levels deep. Accessing a value on the 80x87 stack is much faster than accessing a variable in memory; so to achieve the best possible performance, Turbo Pascal uses the 80x87's stack for storing temporary results.

In theory, very complicated real-type expressions can cause an 80x87 stack overflow. However, this is not likely to occur, since it would require the expression to generate more than eight temporary results.

A more tangible danger lies in recursive function calls. If such constructs are not coded correctly, they can very well cause an 80x87 stack overflow.

Consider the following procedure that calculates Fibonacci numbers using recursion:

```
function Fib(N: Integer): Extended;
begin
  if N = 0 then
    Fib := 0.0
  else
    if N = 1 then
      Fib := 1.0
    else
  Fib := Fib(N - 1) + Fib(N - 2);
end;
```

A call to this version of *Fib* will cause an 80x87 stack overflow for values of *N* larger than 8. The reason is that the calculation of the last assignment requires a temporary on the 80x87 stack to store the result of *Fib(N-1)*. Each recursive invocation allocates one such temporary, causing an overflow the ninth time. The correct construct in this case is

```
function Fib(N: Integer): Extended;
var
  F1, F2: Extended;

begin
  if N = 0 then
    Fib := 0.0
  else
    if N = 1 then
      Fib := 1.0
    else
    begin
      F1 := Fib(N - 1);
      F2 := Fib(N - 2);
      Fib := F1 + F2;
    end;
end;
```

The temporary results are now stored in variables allocated on the 8086 stack. (The 8086 stack can of course also overflow, but this would typically require significantly more recursive calls.)

Writing reals with the 80x87

In the {**$N+**} state, the *Write* and *Writeln* standard procedures output four digits, not two, for the exponent in a floating-point

decimal string to provide for the extended numeric range. Likewise, the *Str* standard procedure returns a four-digit exponent when floating-point format is selected.

Units using the 80x87

Units that use the 80x87 can only be used by other units or programs that are compiled in the {**$N+**} state.

The fact that a unit uses the 80x87 is determined by whether it contains 80x87 instructions—not by the state of the **$N** compiler directive at the time of its compilation. This makes the compiler more forgiving in cases where you accidentally compile a unit (that doesn't use the 80x87) in the {**$N+**} state.

When you compile in numeric processing mode ({**$N+**}), the return values of the floating-point routines in the *System* unit—*Sqrt*, *Pi*, *Sin*, and so on—are of type Extended instead of Real.

Detecting the 80x87

The Windows environment and the WIN87EM.DLL emulator library automatically detect the presence of an 80x87 chip. If an 80x87 is available in your system, it is used. If it isn't, WIN87EM.DLL emulates it in software. You can use the *GetWinFlags* function (defined in the *WinProcs* unit) and the *wf_80x87* bit mask (defined in the *WinTypes* unit) to determine whether an 80x87 processor is present in your system, for example

```
if GetWinFlags and wf_80x87 <> 0 then
  WriteLn('80x87 is present') else
  WriteLn('80x87 is not present');
```

Emulation in assembly language

When linking in object files using {**$L** *filename*} directives, make sure that these object files were compiled with the 80x87 emulation enabled. For example, if you are using 80x87 instructions in assembly language **external** procedures, make sure to enable emulation when you assemble the .ASM files into .OBJ files. Otherwise, the 80x87 instructions cannot be emulated on machines without an 80x87. Use Turbo Assembler's **/E** command-line switch to enable emulation.

PART 3

Inside Turbo Pascal

C H A P T E R

16

Memory issues

This chapter describes in detail the ways Turbo Pascal programs use memory. We'll look the attributes of code segments, the automatic data segment, internal data formats, the heap manager, and direct memory access.

Code segments

Each module (the main program or library and each unit) in a Turbo Pascal application or DLL has its own code segment. The size of a single code segment cannot exceed 64K, but the total size of the code is limited only by the available memory.

Segment attributes

Each code segment has a set of attributes that determine the behavior of the code segment when it is loaded into memory.

MOVEABLE or FIXED

When a code segment is MOVEABLE, Windows can move the segment around in physical memory in order to satisfy other memory allocation requests. When a code segment is FIXED, it *never* moves in physical memory. The preferred attribute is MOVEABLE, and unless it is absolutely necessary to keep a code segment at the same address in physical memory (such as if it contains an interrupt handler), you should use the MOVEABLE attribute.

PRELOAD or DEMANDLOAD

A code segment that has the PRELOAD attribute is automatically loaded when the application or library is activated. The DEMANDLOAD attribute delays the loading of the segment until a routine in the segment is actually called.

DISCARDABLE or PERMANENT

When a segment is DISCARDABLE, Windows can free the memory occupied by the segment when it needs to allocate additional memory. When a segment is PERMANENT, it is kept in memory at all times. When an application makes a call to a DISCARDABLE segment that is not in memory, Windows first loads it from the .EXE file. This takes longer than if the segment was PERMANENT, but it allows an application to execute in less space.

Roughly speaking, a DISCARDABLE segment in a Windows application is much like an overlaid segment in a DOS program, while a PERMANENT segment in a Windows application is like that of a segment that is not overlaid in a DOS program.

Changing attributes

The default attributes of a code segment are MOVEABLE, PRELOAD, and PERMANENT, but you can change this with a **$C** compiler directive. For example,

```
{$C MOVEABLE DEMANDLOAD DISCARDABLE}
```

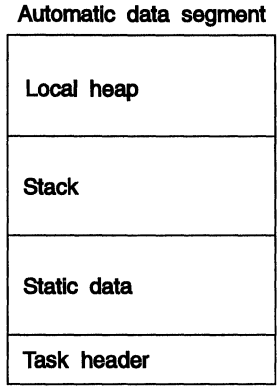
*For details about the **$C** compiler directive, see Chapter 21, "Compiler directives."*

There is no need for a separate overlay manager in a Windows application. The Windows memory manager includes a full set of overlay management services, controlled through code segment attributes. These services are available to any Windows application.

The automatic data segment

Each application or library has one data segment called the "automatic data segment", which can be up to 64K in size. The automatic data segment is always pointed to by the data segment register (DS). It is divided into four sections:

Figure 16.1
Automatic data segment



The first 16 bytes of the automatic data segment always contain the *task header* in which Windows stores various system information.

The *static data* area contains all global variables and typed constants declared by the application or library.

The *stack* is used to store local variables allocated by procedures and functions. On entry to an application, the stack segment register (SS) and the stack pointer (SP) are loaded so that SS:SP points to the first byte past the stack area in the automatic data segment. When procedures and functions are called, SP is moved down to allocate space for parameters, the return address, and local variables. When a routine returns, the process is reversed by incrementing SP to the value it had before the call. The default size of the stack area in the automatic data segment is 8K, but this can be changed with a **$M** compiler directive.

Unlike an application, a library has no stack area in its automatic data segment. When a call is made to a procedure or function in a DLL, the DS register points to the library's automatic data segment, but the SS:SP register pair is not modified. Therefore, a library always uses the stack of the calling application.

The last section in the automatic data segment is the *local heap*. It contains all local dynamic data that was allocated using the *LocalAlloc* function in Windows. The default size of the local heap section is 8K, but this can be changed with a **$M** compiler directive.

Windows allows the automatic data segment to be *movable*, but Turbo Pascal for Windows doesn't support this. The automatic data segment of a Turbo Pascal application or library is always *locked*, thereby ensuring that the selector (segment address) of the automatic data segment never changes. This has no adverse effects when running in standard or extended mode, since a segment retains the same selector even when it is moved in physical memory. In real mode, however, if Windows is required to expand the local heap, it probably won't be able to do so, since the automatic data segment can't be moved. If your application uses the local heap and must run in real mode, you should make sure that the initial size of the local heap (set with a **$M** directive) is large enough to accomodate all local heap allocations.

The heap manager

Windows supports dynamic memory allocations on two different heaps: The *global heap* and the *local heap*.

See the Windows Programming Guide for more details on the global and local heaps.

The global heap is a pool of memory available to all applications. Although global memory blocks of any size can be allocated, the global heap is intended only for "large" memory blocks (256 bytes or more). Each global memory block carries an overhead of at least 20 bytes, and under the Windows standard and 386 enhanced modes, there is a system-wide limit of 8192 global memory blocks, only some of which are available to any given application.

The local heap is a pool of memory available only to your application or library. It exists in the upper part of an application's or library's data segment. The total size of local memory blocks that can be allocated on the local heap is 64K minus the size of the application's stack and static data. For this reason, the local heap is best suited for "small" memory blocks (256 bytes or less). The default size of the local heap is 8K, but you can change this with the **$M** compiler directive.

Turbo Pascal for Windows doesn't support the Mark and Release allocation scheme provided in DOS verions.

Turbo Pascal for Windows includes a *heap manager* which implements the *New*, *Dispose*, *GetMem*, and *FreeMem* standard procedures. The heap manager uses the global heap for all allocations. Since the global heap has a system-wide limit of 8192 memory blocks (which certainly is less than what some applications may require), Turbo Pascal's heap manager includes

a *segment sub-allocator* algorithm to enhance performance and allow a substantially larger number of blocks to be allocated.

To read more about using the heap manager in a DLL, see page 136 in Chapter 10, "Dynamic-link libraries."

This is how the segment sub-allocator works: When allocating a "large" block, the heap manager simply allocates a global memory block using the Windows *GlobalAlloc* routine. When allocating a "small" block, the heap manager allocates a larger global memory block and then divides (sub-allocates) that block into smaller blocks as required. Allocations of "small" blocks reuse all available sub-allocation space before the heap manager allocates a new global memory block, which, in turn, is further sub-allocated.

The *HeapLimit* variable defines the threshold between "small" and "large" heap blocks. The default value is 1024 bytes. The *HeapBlock* variable defines the size the heap manager uses when allocating blocks to be assigned to the sub-allocator. The default value of *HeapBlock* is 8192 bytes. You should have no reason to change the values of *HeapLimit* and *HeapBlock*, but should you decide to do so, make sure that *HeapBlock* is at least four times the size of *HeapLimit*.

Global memory blocks allocated by the heap manager are always fixed; that is, they are always allocated with the *GMEM_FIXED* attribute. This ensures that the selectors (segment addresses) of the blocks don't change. In Windows standard and 386 enhanced modes, fixed blocks can still be moved around in physical memory to make room for other memory allocation requests, so there is no performance penalty associated with using the Turbo Pascal heap manager. In Windows real mode, however, a fixed block must remain fixed in physical memory. This precludes the Windows memory manager from moving it in order to allocate other blocks. If your application is to run in real mode, you may want to consider using the memory management services provided by Windows when allocating dynamic memory blocks.

The HeapError variable

The *HeapError* variables allows you to install a heap error function, which gets called whenever the heap manager cannot complete an allocation request. *HeapError* is a pointer that points to a function with this header:

```
function HeapFunc (Size: Word): Integer; far;
```

Note that the **far** directive forces the heap error function to use the FAR call model.

The heap error function is installed by assigning its address to the *HeapError* variable:

```
HeapError := @HeapFunc;
```

The heap error function gets called whenever a call to *New* or *GetMem* cannot complete the request. The *Size* parameter contains the size of the block that could not be allocated, and the heap error function should attempt to free a block of a least that size.

Before calling the heap error function, the heap manager attempts to allocate the block within its sub-allocation free space as well as through a direct call to the Windows *GlobalAlloc* function.

Depending on its success, the heap error function should return 0, 1, or 2. A return of 0 indicates failure, causing a run-time error to occur immediately. A return of 1 also indicates failure, but instead of a run-time error, it causes *New* or *GetMem* to return a **nil** pointer. Finally, a return of 2 indicates success and causes a retry (which could also cause another call to the heap error function).

The standard heap error function always returns 0, thus causing a run-time error whenever a call to *New* or *GetMem* cannot be completed. However, for many applications, the simple heap error function that follows is more appropriate:

```
function HeapFunc(Size: Word): Integer; far;
begin
  HeapFunc := 1;
end;
```

When installed, this function causes *New* or *GetMem* to return **nil** when they cannot complete the request, instead of aborting the program.

Internal data formats

Integer types

The format selected to represent an integer-type variable depends on its minimum and maximum bounds:

- If both bounds are within the range –128..127 (Shortint), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (byte), the variable is stored as an unsigned byte.
- If both bounds are within the range –32768..32767 (Integer), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (Word), the variable is stored as an unsigned word.
- Otherwise, the variable is stored as a signed double word (Longint).

Char types

A Char, or a subrange of a Char type, is stored as an unsigned byte.

Boolean, WordBool, and LongBool types

A Boolean type is stored as a Byte, a WordBool type is stored as a Word, and a LongBool is stored as a Longint.

Boolean, WordBool, and LongBool types can assume the value of 0 (False) or non-zero (True).

In DOS versions of Turbo Pascal, only a Boolean type existed and its value could be only 0 (False) or 1 (True). Because of this difference, a Turbo Pascal for Windows program can read boolean fields from DOS Turbo Pascal data files, but a DOS-based Turbo Pascal programs will not be able to read Turbo Pascal for Windows data files correctly.

Enumerated types

An enumerated type is stored as an unsigned byte if the enumeration has 256 or fewer values; otherwise, it is stored as an unsigned word.

Floating-point types

The floating-point types (Real, Single, Double, Extended, and Comp) store the binary representations of a sign (+ or –), an *exponent*, and a *significand*. A represented number has the value

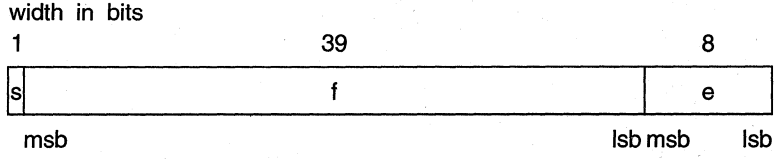
$$+/- \text{ significand} \times 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary decimal point (that is, 0 <= significand < 2).

In the figures that follow, *msb* means most significant bit, and *lsb* means least significant bit. The leftmost items are stored at the highest addresses. For example, for a real-type value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real type

A 6-byte (48-bit) *Real* number is divided into three fields:



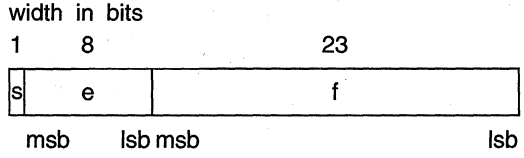
The value *v* of the number is determined by

```
if 0 < e <= 255, then v = (-1)^s * 2^(e-129) * (1.f).
if e = 0, then v = 0.
```

The Real type cannot store denormals, NaNs, and infinities. Denormals become zero when stored in a Real, and NaNs and infinities produce an overflow error if an attempt is made to store them in a Real.

The Single type

A 4-byte (32-bit) Single number is divided into three fields:

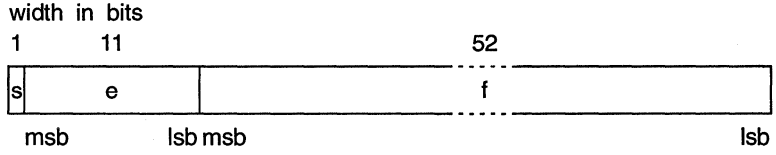


The value *v* of the number is determined by

```
if 0 < e < 255,          then v = (-1)^s * 2^(e-127) * (1.f).
if e = 0   and f <> 0,   then v = (-1)^s * 2^(-126) * (0.f).
if e = 0   and f = 0,    then v = (-1)^s * 0.
if e = 255 and f = 0,    then v = (-1)^s * Inf.
if e = 255 and f <> 0,   then v is a NaN.
```

The Double type

An 8-byte (64-bit) Double number is divided into three fields:

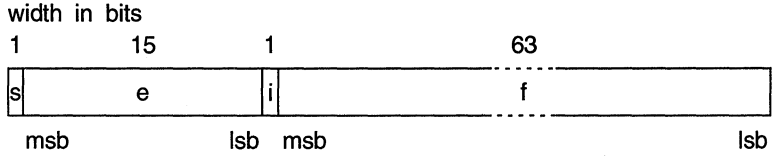


The value *v* of the number is determined by

```
if 0 < e < 2047,         then v = (-1)^s * 2^(e-1023) * (1.f).
if e = 0   and f <> 0,   then v = (-1)^s * 2^(-1022) * (0.f).
if e = 0   and f = 0,    then v = (-1)^s * 0.
if e = 2047 and f = 0,   then v = (-1)^s * Inf.
if e = 2047 and f <> 0,  then v is a NaN.
```

The Extended type

A 10-byte (80-bit) Extended number is divided into four fields:

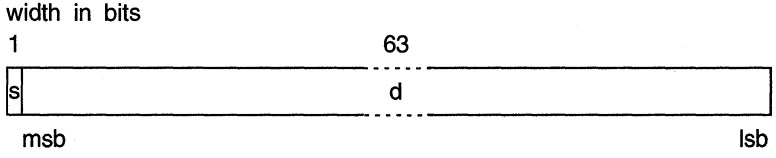


The value *v* of the number is determined by

```
if 0 <= e < 32767,       then v = (-1)^s * 2^(e-16383) * (i.f).
if e = 32767 and f = 0,  then v = (-1)^s * Inf.
if e = 32767 and f <> 0, then v is a NaN.
```

The Comp type

An 8-byte (64-bit) Comp number is divided into two fields:



The value *v* of the number is determined by

```
if s = 1 and d = 0, then v is a NaN
```

Otherwise, *v* is the two's complement 64-bit value.

Pointer types

A Pointer type is stored as a double word, with the offset part in the low word and the segment part in the high word. The pointer value **nil** is stored as a double-word zero.

String types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string. The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (**string**[255]).

Set types

A set is a bit array, where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is calculated as

```
ByteSize = (Max div 8) - (Min div 8) + 1
```

where *Min* and *Max* are the lower and upper bounds of the base type of that set. The byte number of a specific element *E* is

```
ByteNumber = (E div 8) - (Min div 8)
```

and the bit number within that byte is

```
BitNumber = E mod 8
```

where *E* denotes the ordinal value of the element.

Array types

An array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multi-dimensional array is stored with the rightmost dimension increasing first.

Record types

The fields of a record are stored as a contiguous sequence of variables. The first field is stored at the lowest memory address. If the record contains variant parts, then each variant starts at the same memory address.

File types

File types are represented as records. Typed files and untyped files occupy 128 bytes, which are laid out as follows:

```
type
  TFileRec = record
    Handle: Word;
    Mode: Word;
    RecSize: Word;
    Private: array[1..26] of Byte;
    UserData: array[1..16] of Byte;
    Name: array[0..79] of Char;
  end;
```

Text files occupy 256 bytes, which are laid out as follows:

```
type
  TTextBuf = array[0..127] of Char;
  TTextRec = record
    Handle: Word;
    Mode: Word;
    BufSize: Word;
    Private: Word;
    BufPos: Word;
    BufEnd: Word;
    BufPtr: ^TTextBuf;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..16] of Byte;
    Name: array[0..79] of Char;
    Buffer: TTextBuf;
  end;
```

Handle contains the file's handle (when open) as returned by DOS.

The *Mode* field can assume one of the following "magic" values:

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

fmClosed indicates that the file is closed. *fmInput* and *fmOutput* indicate that the file is a text file that has been reset (*fmInput*) or rewritten (*fmOutput*). *fmInOut* indicates that the file variable is a typed or an untyped file that has been reset or rewritten. Any other value indicates that the file variable has not been assigned (and thereby not initialized).

The *UserData* field is never accessed by Turbo Pascal, and is free for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file. The section entitled "Text file device drivers" in Chapter 19 provides information on that subject.

Procedural types

A procedural type is stored as a double word, with the offset part of the referenced procedure in the low word and the segment part in the high word.

Direct memory access

Turbo Pascal implements three predefined arrays, *Mem*, *MemW*, and *MemL*, which are used to directly access memory. Each component of *Mem* is a byte, each component of *MemW* is a Word, and each component of *MemL* is a Longint.

The *Mem* arrays use a special syntax for indexes: Two expressions of the integer type Word, separated by a colon, are used to specify

the segment base and offset of the memory location to access. Some examples include

```
Mem[$0040:$0049] := 7;
Data := MemW[Seg(V):Ofs(V)];
MemLong := MemL[64:3*4];
```

The first statement stores the value 7 in the byte at $0040:$0049. The second statement moves the Word value stored in the first 2 bytes of the variable *V* into the variable *Data*. The third statement moves the Longint value stored at $0040:$000C into the variable *MemLong*.

Although Turbo Pascal permits you to access memory directly, this is not a safe practice under Windows. You should allow Windows to manage memory issues for you or your programs may crash.

C H A P T E R

17

Objects

Internal data format of objects

The internal data format of an object resembles that of a record. The fields of an object are stored in order of declaration, as a contiguous sequence of variables. Any fields inherited from an ancestor type are stored before the new fields defined in the descendant type.

If an object type defines virtual methods, constructors, or destructors, the compiler allocates an extra field in the object type. This 16-bit field, called the *virtual method table (VMT) field,* is used to store the offset of the object type's VMT in the data segment. The VMT field immediately follows after the ordinary fields in the object type. When an object type inherits virtual methods, constructors, or destructors, it also inherits a VMT field, so an additional one is not allocated.

Initialization of the VMT field of an instance is handled by the object type's constructor(s). A program never explicitly initializes or accesses the VMT field.

The following examples illustrate the internal data formats of object types:

```
type
  PLocation = ^TLocation;
  TLocation = object
    X, Y: Integer;
```

```
  procedure Init(PX, PY: Integer);
  function GetX: Integer;
  function GetY: Integer;
end;

PPoint = ^TPoint;
TPoint = object(TLocation)
  Color: Integer;
  constructor Init(PX, PY, PColor: Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure MoveTo(PX, PY: Integer); virtual;
end;

PCircle = ^TCircle;
TCircle = object(TPoint)
  Radius: Integer;
  constructor Init(PX, PY, PColor, PRadius: Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure Fill; virtual;
end.
```

Figure 17.1 shows layouts of instances of *TLocation*, *TPoint*, and *TCircle*; each box corresponds to one word of storage.

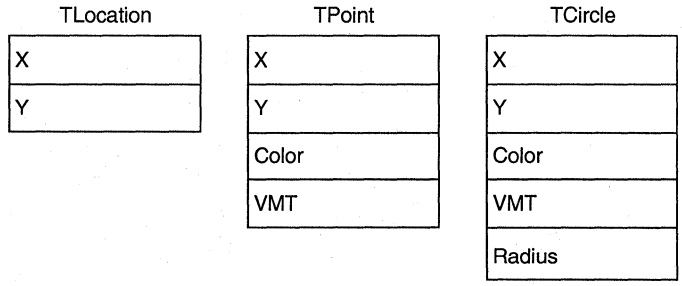


Figure 17.1
Layouts of instances of TLocation, TPoint, and TCircle

Virtual method tables

Each object type that contains or inherits virtual methods, constructors, or destructors has a VMT associated with it, which is stored in the initialized part of the program's data segment. There is only one VMT per object type (not one per instance), but two distinct object types never share a VMT, no matter how identical they appear to be. VMTs are built automatically by the compiler, and are never directly manipulated by a program. Likewise, pointers to VMTs are automatically stored in object type instances

by the object type's constructor(s) and are never directly manipulated by a program.

The first word of a VMT contains the size of instances of the associated object type; this information is used by constructors and destructors to determine how many bytes to allocate or dispose of, using the extended syntax of the *New* and *Dispose* standard procedures.

The second word of a VMT contains the negative size of instances of the associated object type; this information is used by the virtual method call validation mechanism to detect uninitialized objects (instances for which no constructor call has been made), and to check the consistency of the VMT. When virtual call validation is enabled (using the {**$R+**} compiler directive, which has been expanded to include virtual method checking), the compiler generates a call to a VMT validation routine before each virtual call. The VMT validation routine checks that the first word of the VMT is not zero, and that the sum of the first and the second word is zero. If either check fails, run-time error 210 is generated.

Enabling range-checking and virtual method call checking slows down your program and makes it somewhat larger, so use the {**R+**} state only when debugging, and switch to the {**$R-**} state for the final version of the program.

The third word of a VMT contains the data segment offset of the object type's DMT (Dynamic Method Table), or zero if the object type has no dynamic methods.

The fourth word of a VMT is reserved, and always contains zero.

Finally, starting at offset 8 in the VMT, comes a list of 32-bit method pointers, one per virtual method in the object type, in order of declaration. Each slot contains the address of the corresponding virtual method's entry point.

Figure 17.2 shows the layouts of the VMTs of the *TPoint* and *TCircle* types; each small box corresponds to one word of storage, and each large box corresponds to two words of storage.

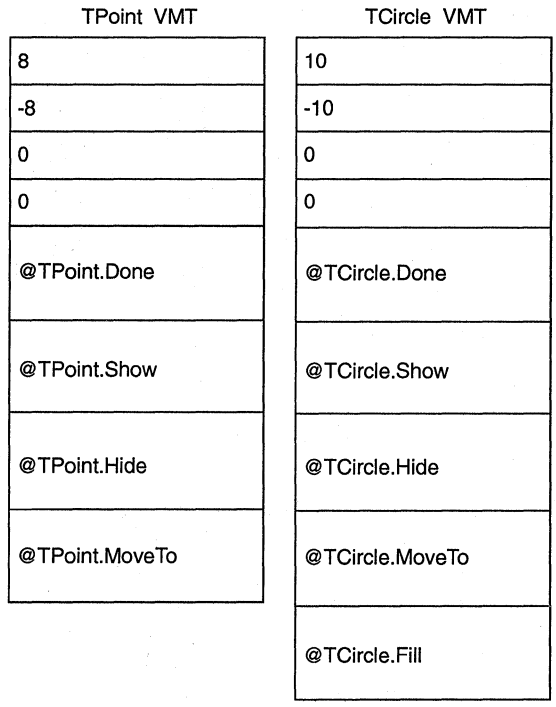


Figure 17.2
TPoint and TCircle's VMT layouts

Notice how *TCircle* inherits the *Done*, and *MoveTo* methods from *TPoint*, and how it overrides the *Show* and *Hide* methods.

As mentioned already, an object type's constructors contain special code that stores the offset of the object type's VMT in the instance being initialized. For example, given an instance *P* of type *Pointer*, and an instance *C* of type *TCircle*, a call to *P.Init* will automatically store the offset of *TPoint*'s VMT in *P*'s VMT field, and a call to *C.Init* will likewise store the offset of *TCircle*'s VMT in *C*'s VMT field. This automatic initialization is part of a constructor's entry code, so when control arrives at the **begin** of the constructor's statement part, the VMT field *Self* will already have been set up. Thus, if the need arises, a constructor can make calls to virtual methods.

Dynamic method tables

The VMT for an object type contains a four-byte entry (a method pointer) for each virtual method declared in the object type and any of its ancestors. In cases where ancestral type(s) define a large number of virtual methods, the process of creating derived types can use up quite a lot of memory, especially if many derived types are created. Even though the derived types may override only a few of the inherited methods, the VMT of each derived type contains method pointers for all inherited virtual methods, even if they haven't changed.

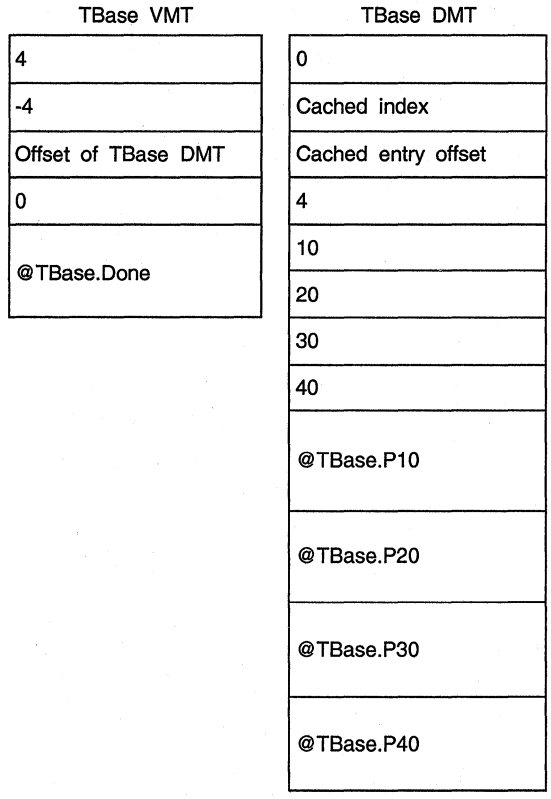
Dynamic methods provide an alternative in such situations. Turbo Pascal for Windows introduces a new method table format and a new way of dispatching late-bound method calls. Instead of encoding a pointer for *all* late-bound methods in an object type, a DMT (Dynamic Method Table) encodes only the methods that were *overridden* in the object type. When descendant types override only a few of a large number of inherited late-bound methods, the dynamic method table format uses less space than the format used by VMTs.

The following two object types illustrate DMT formats:

```
type
  TBase = object
    X: Integer;
    constructor Init;
    destructor Done; virtual;
    procedure P10; virtual 10;
    procedure P20; virtual 20;
    procedure P30; virtual 30;
    procedure P40; virtual 40;
  end;

type
  TDerived = object(TBase)
    Y: Integer;
    constructor Init;
    destructor Done; virtual;
    procedure P10; virtual 10;
    procedure P30; virtual 30;
    procedure P50; virtual 50;
  end;
```

Figures 17.3 and 17.4 shows the layouts of the VMTs and DMTs of *TBase* and *TDerived*. Each small box corresponds to one word of storage, and each large box corresponds to two words of storage.

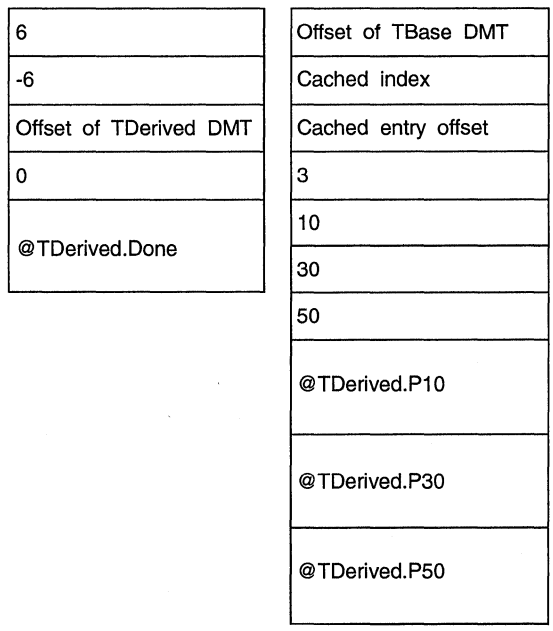
Figure 17.3
TBase's VMT and DMT layouts



An object type has a DMT only if it introduces or overrides dynamic methods. If an object type inherits dynamic methods, but doesn't override any of them or introduce new ones, it simply inherits the DMT of its ancestor.

As is the case for VMTs, DMTs are stored in the initialized part of the application's data segment.

Figure 17.4
TDerived's VMT and DMT layouts



The first word of a DMT contains the data segment offset of the *parent DMT*, or zero if there is no parent DMT.

The second and third words of a DMT are used to cache dynamic method lookups, as is described on page 213.

The fourth word of a DMT contains the *DMT entry count*. It is immediately followed by a list of words, each of which contain a dynamic method index, and then followed by a list of corresponding method pointers. The length of each list is given by the DMT entry count.

The SizeOf function

When applied to an instance of an object type that has a VMT, *SizeOf* returns the size stored in the VMT. Thus, for object types that have a VMT, *SizeOf* always returns the *actual* size of the instance, rather than the *declared* size.

The TypeOf function

Turbo Pascal's new standard function *TypeOf* returns a pointer to an object type's VMT. *TypeOf* takes a single parameter, which can be either an object type identifier or an object type instance. In both cases, the result, which is of type *Pointer*, is a pointer to the object type's VMT. *TypeOf* can be applied only to object types that have a VMT—all other types result in an error.

The *TypeOf* function can be used to test the actual type of an instance. For example,

```
if TypeOf(Self) = TypeOf(Point) then ...
```

Virtual method calls

To call a virtual method, the compiler generates code that picks up the VMT address from the VMT field in the object, and then calls via the slot associated with the method. For example, given a variable *PP* of type *PointPtr*, the call *PP^.Show* generates the following code:

```
les     DI,EP               ;Load EP into ES:DI
push    ES                  ;Pass as Self parameter
push    DI
mov     DI,ES:[DI+6]        ;Pick up VMT offset from VMT field
call    DWORD PTR [DI+12]   ;Call VMT entry for Show
```

The type compatibility rules of object types allow *PP* to point at a *Point* or a *TCircle*, or at any other descendant of *TPoint*. And if you examine the VMTs shown here, you'll see that for a *TPoint*, the entry at offset 12 in the VMT points to *TPoint.Show*; whereas for a *TCircle*, it points to *TCircle.Show*. Thus, depending upon the *actual* run-time type of *PP*, the **CALL** instruction calls *TPoint.Show* or *TCircle.Show*, or the *Show* method of any other descendant of *TPoint*.

If *Show* had been a static method, this code would have been generated for the call to *PP^.Show*:

```
les     di,EP               ;Load EP into ES:DI
push    es                  ;Pass as Self parameter
push    di
call    TPoint.Show          ;Directly call TPoint.Show
```

Here, no matter what *PP* points to, the code will always call the *TPoint.Show* method.

Dynamic method calls

Dispatching a dynamic method call is somewhat more complicated and time consuming than dispatching a virtual method call. Instead of using a CALL instruction to call through a method pointer at a static offset in the VMT, the object type's DMT and parent DMTs must be *scanned* to find the "topmost" occurrence of a particular dynamic method index, and then a call must be made through the corresponding method pointer. This process involves far more instructions than can be coded "in-line", so the Turbo Pascal RTL contains a dispatch support routine which is used when making dynamic method calls.

Had the *Show* method of the preceeding type *TPoint* been declared as a dynamic method (with a dynamic method index of 200), the call *PP^.Show*, where *PP* is of type *TPointPtr*, would generate the following code:

```
les     di,PP             ;Load PP into ES:DI
push    es                ;Pass as Self parameter
push    di
mov     di,es:[di+6]      ;Pick up VMT offset from VMT field
mov     ax,200            ;Load dynamic method index into AX
call    Dispatch          ;Call RTL routine to dispatch call
```

The RTL dispatcher first picks up the DMT offset from the VMT pointed to by the DI register. Then, using the "cached index" field of the DMT, the dispatcher checks if the dynamic method index of the method being called is the same as the last one that was called. If so, it immediately transfers control to the method, by jumping indirectly through the method pointer stored at the offset given by the "cached entry offset" field. If the dynamic index of the method being called is not the same as the one stored in the cache, the dispatcher scans the DMT and the parent DMTs (by following the parent links in the DMTs) until it locates an entry with the given dynamic method index. The index and the offset of the corresponding method pointer is then stored in the DMT's cache fields, and control is transferred to the method. If, for some reason, the dispatcher cannot find an entry with the given dynamic method index, indicating that the DMTs have somehow been trashed, it terminates the application with a run-time error 210.

In spite of caching and a highly optimized RTL dispatch support routine, the dispatching of a dynamic method call takes

substantially longer than a virtual method call. However, in cases where the actions performed by the dynamic methods themselves take up a lot of time, the the amount space saved by using DMTs may well outweigh this penalty.

Method calling conventions

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter, *Self*, that corresponds to a **var** parameter of the same type as the method's object type. The *Self* parameter is always passed as the last parameter, and always takes the form of a 32-bit pointer to the instance through which the method is called. For example, given a variable *PP* of type *TPointPtr* as defined earlier, the call *PP*^.*MoveTo*(10, 20) is coded as follows:

```
mov     ax,10               ;Load 10 into AX
push    ax                  ;Pass as Days parameter
mov     ax,20               ;Load 20 into AX
push    ax                  ;Pass as PY parameter
les     di,PP               ;Load PP into ES:DI
push    es                  ;Pass as Self parameter
push    di
mov     di,es:[di+6]        ;Pick up VMT offset from VMT field
call    DWORD PTR [di+20]   ;Call VMT entry for MoveTo
```

Upon returning, a method must remove the *Self* parameter from the stack, just as it must remove any normal parameters.

Methods always use the far call model, regardless of the setting of the **$F** compiler directive.

Constructors and destructors

Constructors and destructors use the same calling conventions as normal methods, except that an additional word-sized parameter, called the *VMT* parameter, is passed on the stack just before the *Self* parameter.

For constructors, the VMT parameter contains the VMT offset to store in *Self*'s VMT field in order to initialize *Self*.

Furthermore, when a constructor is called to allocate a dynamic object, using the extended syntax of the *New* standard procedure, a **nil** pointer is passed in the *Self* parameter. This causes the con-

structor to allocate a new dynamic object, the address of which is passed back to the caller in DX:AX when the constructor returns. If the constructor could not allocate the object, a **nil** pointer is returned in DX:AX.

See "Constructor error recovery" on page 220.

Finally, when a constructor is called using a qualified method identifier (that is, an object type identifier), followed by a period and a method identifier, a value of zero is passed in the VMT parameter. This indicates to the constructor that it should *not* initialize the VMT field of *Self*.

For destructors, a 0 in the VMT parameter indicates a normal call, and a nonzero value indicates that the destructor was called using the extended syntax of the *Dispose* standard procedure. This causes the destructor to deallocate *Self* just before returning (the size of *Self* is found by looking at the first word of *Self*'s VMT).

Extensions to New and Dispose

The *New* and *Dispose* standard procedures have been extended to allow a constructor call or destructor call as a second parameter for allocating or disposing a dynamic object type variable. The syntax is

```
New(P, Construct)
```

and

```
Dispose(P, Destruct)
```

where *P* is a pointer variable, pointing to an object type, and *Construct* and *Destruct* are calls to constructors and destructors of that object type. For *New*, the effect of the extended syntax is the same as executing

```
New(P);
P^.Construct;
```

And for *Dispose*, the effect of the extended syntax is the same as executing

```
P^.Destruct;
Dispose(P);
```

Without the extended syntax, occurrences of such "pairs" of a call to *New* followed by a constructor call, and a destructor call followed by a call to *Dispose* would be very common. The extended syntax improves readability, and also generates shorter and more efficient code.

The following illustrates the use of the extended *New* and *Dispose* syntax:

```
var
  SP: StrFieldPtr;
  ZP: ZipFieldPtr;
begin
  New(SP, Init(1, 1, 25, 'Firstname'));
  New(ZP, Init(1, 2, 5, 'Zip code', 0, 99999));
  SP^.Edit;
  ZP^.Edit;
  ...
  Dispose(ZP, Done);
  Dispose(SP, Done);
end;
```

An additional extension allows *New* to be used as a *function*, which allocates and returns a dynamic variable of a specified type. The syntax is

```
New(T)
```

or

```
New(T, Construct)
```

In the first form, *T* can be any pointer type. In the second form, *T* must point to an object type, and *Construct* must be a call to a constructor of that object type. In both cases, the type of the function result is *T*.

Here's an example:

```
var
  F1, F2: FieldPtr;
begin
  F1 := New(StrFieldPtr, Init(1, 1, 25, 'Firstname'));
  F2 := New(ZipFieldPtr, Init(1, 2, 5, 'Zip code', 0, 99999));
  ...
  WriteLn(F1^.GetStr);                    { Calls StrField.GetStr }
  WriteLn(F2^.GetStr);                    { Calls ZipField.GetStr }
  ...
  Dispose(F2, Done);                          { Calls Field.Done }
  Dispose(F1, Done);                        { Calls StrField.Done }
end;
```

Notice that even though *F1* and *F2* are of type *FieldPtr*, the extended pointer assignment compatibility rules allow *F1* and *F2* to be assigned a pointer to any descendant of *Field*; and since *GetStr* and *Done* are virtual methods, the virtual method dispatch

mechanism correctly calls *StrField.GetStr*, *ZipField.GetStr*, *Field.Done*, and *StrField.Done*, respectively.

Assembly language methods

Method implementations written in assembly language can be linked with Turbo Pascal programs using the **$L** compiler directive and the **external** reserved word. The declaration of an external method in an object type is no different than that of a normal method; however, the implementation of the method lists only the method header followed by the reserved word **external**. In an assembly language source text, an @ is used instead of a period (.) to write qualified identifiers (the period already has a different meaning in assembly language and cannot be part of an identifier). For example, the Pascal identifier *Rect.Init* is written as *Rect@Init* in assembly language. The @ syntax can be used to declare both **PUBLIC** and **EXTRN** identifiers.

As an example of assembly language methods, we've implemented a simple *Rect* object.

```
type
  Rect = object
    X1, Y1, X2, Y2: Integer;
    procedure Init(XA, YA, XB, YB: Integer);
    procedure Union(var R: Rect);
    function Contains(X, Y: Integer): Boolean;
  end;
```

A *Rect* represents a rectangle bounded by four coordinates, *X1*, *Y1*, *X2*, and *Y2*. The upper left corner of a rectangle is defined by *X1* and *Y1*, and the lower right corner is defined by *X2* and *Y2*. The *Init* method assigns values to the rectangle's coordinates; the *Union* method calculates the smallest rectangle that contains both the rectangle itself and another rectangle; and the *Contains* method returns True if a given point is within the rectangle, or False if not. Other methods, such as moving, resizing, calculating intersections, and testing for equality, could easily be implemented to make *Rect* a more useful object.

The Pascal implementations of *Rect*'s methods list only the method header followed by an **external** reserved word.

```
{$L RECT}

procedure Rect.Init(XA, YA, XB, YB: Integer); external;
```

```
procedure Rect.Union(var R: Rect); external;
function Rect.Contains(X, Y: Integer): Boolean; external;
```

There is, of course, no requirement that all methods be implemented as externals. Each individual method can be implemented in either Pascal or in assembly language, as desired.

The assembly language source file, RECT.ASM, that implements the three external methods is listed here.

```
        TITLE   Rect
        LOCALS  @@

; Rect structure

Rect    STRUC
X1      DW      ?
Y1      DW      ?
X2      DW      ?
Y2      DW      ?
Rect    ENDS

Code    SEGMENT BYTE PUBLIC

        ASSUME  cs:code

; Procedure Rect.Init(XA, YA, XB, YB: Integer)

        PUBLIC  Rect@Init

Rect@Init       PROC    FAR

@XA             EQU     (WORD PTR [bp+16])
@YA             EQU     (WORD PTR [bp+14])
@XB             EQU     (WORD PTR [bp+12])
@YB             EQU     (WORD PTR [bp+10])
@Self           EQU     (DWORD PTR [bp+6])

        push    bp              ;Save bp
        mov     bp,sp           ;Set up stack frame
        les     di,@Self        ;Load Self into ES:DI
        cld                     ;Move forwards
        mov     ax,@XA          ;X1 := XA
        stosw
        mov     ax,@YA          ;Y1 := YA
        stosw
        mov     ax,@XB          ;X2 := XB
        stosw
        mov     ax,@YB          ;Y2 := YB
        stosw
        pop     bp              ;Restore BP
        ret     12              ;Pop parameters and return

Rect@Init       ENDP
```

```
; Procedure Rect.Union(var R: Rect)

        PUBLIC  Rect@Union

Rect@Union      PROC    FAR

@R              EQU     (DWORD PTR [bp+10])
@Self           EQU     (DWORD PTR [bp+6])

        push    bp              ;Save BP
        mov     bp,sp           ;Set up stack frame
        push    ds              ;Save DS
        lds     si,@R           ;Load R into DS:SI
        les     di,@Self        ;Load Self into ES:DI
        cld                     ;Move forward
        lodsw                   ;If R.X1 >= X1 goto @@1
        scasw
        jge     @@1
        dec     di              ;X1 := R.X1
        dec     di
        stosw
@@1:    lodsw                   ;If R.Y1 >= Y1 goto @@2
        scasw
        jge     @@2
        dec     di              ;Y1 := R.Y1
        dec     di
        stosw
@@2:    lodsw                   ;If R.X2 <= X2 goto @@3
        scasw
        jle     @@3
        dec     di              ;X2 := R.X2
        dec     di
        stosw
@@3:    lodsw                   ;If R.Y2 <= Y2 goto @@4
        scasw
        jle     @@4
        dec     di              ;Y2 := R.Y2
        dec     di
        stosw
@@4:    pop     ds              ;Restore DS
        pop     bp              ;Restore BP
        ret     8               ;Pop parameters and return

Rect@Union      ENDP

; Function Rect.Contains(X, Y: Integer): Boolean

        PUBLIC  Rect@Contains

Rect@Contains   PROC    FAR

@X              EQU     (WORD PTR [bp+12])
@Y              EQU     (WORD PTR [bp+10])
```

```
@Self           EQU     (DWORD PTR [bp+6])

                push    bp              ;Save BP
                mov     bp,sp           ;Set up stack frame
                les     di,@Self        ;Load Self into ES:DI
                mov     al,0            ;Return false
                mov     dx,@X           ;If (X < X1) or (X > X2) goto @@1
                cmp     dx,es:[di].X1
                jl      @@1
                cmp     dx,es:[di].X2
                jg      @@1
                mov     dx,@Y           ;If (Y < Y1) or (Y > Y2) goto @@2
                cmp     dx,es:[di].Y1
                jl      @@1
                cmp     dx,es:[di].Y2
                jg      @@1
                inc     ax              ;Return true
@@1:            pop     bp              ;Restore BP
                ret     8               ;Pop parameters and return

Rect@Contains   ENDP

Code    ENDS

        END
```

Constructor error recovery

As described in Chapter 16, Turbo Pascal allows you to install a heap error function through the *HeapError* variable in the *System* unit. This functionality is still supported in Turbo Pascal, but now it also affects the way object type constructors work.

By default, when there is not enough memory to allocate a dynamic instance of an object type, a constructor call using the extended syntax of the *New* standard procedure generates run-time error 203. If you install a heap error function that returns 1 rather than the standard function result of 0, a constructor call through *New* will return **nil** when it cannot complete the request (instead of aborting the program).

The code that performs allocation and VMT field initialization of a dynamic instance is part of a constructor's entry sequence: When control arrives at the **begin** of the constructor's statement part, the instance will already have been allocated and initialized. If allocation fails, and if the heap error function returns 1, the constructor skips execution of the statement part and returns a **nil** pointer;

thus, the pointer specified in the *New* construct that called the constructor is set to **nil**.

There's a new standard procedure called Fail.

Once control arrives at the **begin** of a constructor's statement part, the object type instance is guaranteed to have been allocated and initialized successfully. However, the constructor itself might attempt to allocate dynamic variables, in order to initialize pointer fields in the instance, and these allocations might in turn fail. If that happens, a well-behaved constructor should reverse any successful allocations, and finally deallocate the object type instance so that the net result becomes a **nil** pointer. To make such "backing out" possible, Turbo Pascal implements a new standard procedure called *Fail*, which takes no parameters and can be called only from within a constructor. A call to *Fail* causes a constructor to deallocate the dynamic instance that was allocated upon entry to the constructor, and causes the return of a **nil** pointer to indicate its failure.

When dynamic instances are allocated through the extended syntax of *New*, a resulting value of **nil** in the specified pointer variable indicates that the operation failed. Unfortunately, there is no such pointer variable to inspect after the construction of a static instance or when an inherited constructor is called. Instead, Turbo Pascal allows a constructor to be used as a Boolean function in an expression: A return value of True indicates success, and a return value of False indicates failure due to a call to *Fail* within the constructor.

The following program implements two simple object types that contain pointers. This first version of the program does not implement constructor error recovery.

```
type
  LinePtr = ^Line;
  Line = string[79];

  BasePtr = ^Base;
  Base = object
    L1, L2: LinePtr;
    constructor Init(S1, S2: Line);
    destructor Done; virtual;
    procedure Dump; virtual;
  end;

  DerivedPtr = ^Derived;
  Derived = object(Base)
    L3, L4: LinePtr;
    constructor Init(S1, S2, S3, S4: Line);
```

```
    destructor Done; virtual;
    procedure Dump; virtual;
  end;

var
  BP: BasePtr;
  DP: DerivedPtr;

constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  L1^ := S1;
  L2^ := S2;
end;

destructor Base.Done;
begin
  Dispose(L2);
  Dispose(L1);
end;

procedure Base.Dump;
begin
  WriteLn('B: ', L1^, ', ', L2^, '.');
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  Base.Init(S1, S2);
  New(L3);
  New(L4);
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  Dispose(L4);
  Dispose(L3);
  Base.Done;
end;

procedure Derived.Dump;
begin
  WriteLn('D: ', L1^, ', ', L2^, ', ', L3^, ', ', L4^, '.');
end;

begin
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('North', 'East', 'South', 'West'));
  BP^.Dump;
  DP^.Dump;
```

```
  Dispose(DP, Done);
  Dispose(BP, Done);
end.
```

The next example demonstrates how the previous one can be rewritten to implement error recovery. The type and variable declarations are not repeated, because they remain the same.

```
constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  if (L1 = nil) or (L2 = nil) then
  begin
    Base.Done;
    Fail;
  end;
  L1^ := S1;
  L2^ := S2;
end;

destructor Base.Done;
begin
  if L2 <> nil then Dispose(L2);
  if L1 <> nil then Dispose(L1);
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  if not Base.Init(S1, S2) then Fail;
  New(L3);
  New(L4);
  if (L3 = nil) or (L4 = nil) then
  begin
    Derived.Done;
    Fail;
  end;
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  if L4 <> nil then Dispose(L4);
  if L3 <> nil then Dispose(L3);
  Base.Done;
end;

{$F+}
function HeapFunc(Size: Word): Integer;
begin
```

```
  HeapFunc := 1;
end;
{$F-}

begin
  HeapError := @HeapFunc;              { Install heap error handler }
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('North', 'East', 'South', 'West'));
  if (BP = nil) or (DP = nil) then
    WriteLn('Allocation error')
  else
  begin
    BP^.Dump;
    DP^.Dump;
  end;
  if DP <> nil then Dispose(DP, Done);
  if BP <> nil then Dispose(BP, Done);
end.
```

Notice how the corresponding destructors in *Base.Init* and *Derived.Init* are used to reverse any successful allocations before *Fail* is called to finally fail the operation. Also notice that in *Derived.Init,* the call to *Base.Init* is coded within an expression so that the success of the inherited constructor can be tested.

C H A P T E R

18

Control issues

This chapter describes in detail the various ways that Turbo Pascal implements program control. Included are calling conventions, exit procedures, interrupt handling and error handling.

Calling conventions

Parameters are transferred to procedures and functions via the stack. Before calling a procedure or function, the parameters are pushed onto the stack in their order of declaration. Before returning, the procedure or function removes all parameters from the stack.

The skeleton code for a procedure or function call looks like this:

```
PUSH   Param1
PUSH   Param2
:
PUSH   ParamX
CALL   ProcOrFunc
```

Parameters are passed either by *reference* or by *value*. When a parameter is passed by reference, a pointer that points to the actual storage location is pushed onto the stack. When a parameter is passed by value, the actual value is pushed onto the stack.

Variable parameters

Variable parameters (**var** parameters) are always passed by reference—a pointer points to the actual storage location.

Value parameters

Value parameters are passed by value or by reference depending on the type and size of the parameter. In general, if the value parameter occupies 1, 2, or 4 bytes, the value is pushed directly onto the stack. Otherwise a pointer to the value is pushed, and the procedure or function then copies the value into a local storage location.

The 8086 does not support byte-sized **PUSH** and **POP** instructions, so byte-sized parameters are always transferred onto the stack as words. The low-order byte of the word contains the value, and the high-order byte is unused (and undefined).

An integer type or parameter is passed as a byte, a word, or a double word, using the same format as an integer-type variable. (For double words, the high-order word is pushed before the low-order word so that the low-order word ends up at the lowest address.)

A Char-type parameter is passed as an unsigned byte.

A Boolean-type parameter is passed as a byte with the value 0 or 1.

An enumerated-type parameter is passed as an unsigned byte if the enumeration has 256 or fewer values; otherwise, it is passed as an unsigned word.

A Real-type parameter (type Real) is passed as 6 bytes on the stack, thus being an exception to the rule that only 1-, 2-, and 4-byte values are passed directly on the stack.

A floating-point type parameter (Real, Single, Double, Extended, and Comp) is passed as 4, 6, 8, or 10 bytes on the stack, thus being an exception to the rule that only 1-, 2-, and 4-byte values are passed directly on the stack.

Version 4.0 of Turbo Pascal passed 80x87-type parameters (Single, Double, Extended, and Comp) on the internal stack of the 80x87 numeric coprocessor. For reasons of compatibility with other

languages, and to avoid 80x87 stack overflows, this version uses the 8086 stack.

A pointer-type parameter is passed as a double word (the segment part is pushed before the offset part so that the offset part ends up at the lowest address).

A string-type parameter is passed as a pointer to the value.

A set-type parameter is passed as a pointer to an "unpacked" set that occupies 32 bytes.

Arrays and records with 1, 2, or 4 bytes are passed directly onto the stack. Other arrays and records are passed as pointers to the value.

Function results

Ordinal-type function results (Integer, Char, Boolean, and enumeration types) are returned in the CPU registers: Bytes are returned in AL, words are returned in AX, and double words are returned in DX:AX (high-order word in DX, low-order word in AX).

Real-type function results (type Real) are returned in the DX:BX:AX registers (high-order word in DX, middle word in BX, low-order word in AX).

80x87-type function results (type Single, Double, Extended, and Comp) are returned in the 80x87 coprocessor's top-of-stack register (ST(0)).

Pointer-type function results are returned in DX:AX (segment part in DX, offset part in AX).

For a string-type function result, the caller pushes a pointer to a temporary storage location before pushing any parameters, and the function returns a string value in that temporary location. The function must not remove the pointer.

NEAR and FAR calls

The 8086 CPU supports two kinds of call and return instructions: near and far. The near instructions transfer control to another location within the same code segment, and the far instructions allow a change of code segment.

A **NEAR CALL** instruction pushes a 16-bit return address (offset only) onto the stack, and a **FAR CALL** instruction pushes a 32-bit return address (both segment and offset). The corresponding **RET** instructions pop only an offset or both an offset and a segment.

Turbo Pascal will automatically select the correct call model based on the procedure's declaration. Procedures declared in the interface section of a unit are far—they can be called from other units. Procedures declared in a program or in the **implementation** section of a unit are near—they can only be called from within that program or unit.

For some specific purposes, a procedure may be required to be far. For example, in an overlaid application, all procedures and functions are generally required to be far; likewise, if a procedure or function is to be assigned to a procedural variable, it has to be far. The **$F** compiler directive is used to override the compiler's automatic call model selection. Procedures and functions compiled in the {**$F+**} state are always far; in the {**$F-**} state, Turbo Pascal automatically selects the correct model. The default state is {**$F-**}.

Nested procedures and functions

A procedure or function is said to be nested when it is declared within another procedure or function. By default, nested procedures and functions always use the near call model, since they are only "visible" within a specific procedure or function in the same code segment. However, in an overlaid application, a {**$F+**} directive is generally used to force all procedures and functions to be far, including those that are nested.

When calling a nested procedure or function, the compiler generates a **PUSH** BP instruction just before the **CALL**, in effect passing the caller's BP as an additional parameter. Once the called procedure has set up its own BP, the caller's BP is accessible as a word stored at [BP + 4], or at [BP + 6] if the procedure is far. Using this link at [BP + 4] or [BP + 6], the called procedure can access the local variables in the caller's stack frame. If the caller itself is also a nested procedure, it also has a link at [BP + 4] or [BP + 6], and so on. The following example demonstrates how to access local variables from an **inline** statement in a nested procedure:

*Nested procedures and functions cannot be declared with the **external** directive, and they cannot be procedural parameters.*

```
procedure PA; near;
var
  IntA: Integer;

procedure B; far;
var
  IntB: Integer;

procedure C; near;
var
  IntC: Integer;
begin
  inline(
    $8B/$46/<IntC/        { MOV  AX,[BP + IntC]    ;AX = IntC }
    $8B/$5E/$04/          { MOV  BX,[BP + 4]       ;BX = B's stack
                                                    frame }
    $36/$8B/$47/<IntB/    { MOV  AX,SS:[BX + IntB] ;AX = IntB }
    $8B/$5E/$04/          { MOV  BX,[BP + 4]       ;BX = B's stack
                                                    frame }
    $36/$8B/$5F/$06/      { MOV  BX,SS:[BX + 6]    ;BX = A's stack
                                                    frame }
    $36/$8B/$47/<IntA);   { MOV  AX,SS:[BX + IntA] ;AX = IntA }
end;

begin end;

begin end;
```

Entry and exit code

The standard entry and exit code for a procedure or function using the near call model is:

```
push    BP              ;Save BP
mov     BP,SP           ;Set up stack frame
sub     SP,LocalSize    ;Allocate locals (if any)
.
.
mov     SP,BP           ;Deallocate locals (if any)
pop     BP              ;Restore BP
retn    ParamSize       ;Remove parameters and return
```

For information on using exit procedures in a DLL, see Chapter 10, "Dynamic-link libraries".

The entry and exit code for a routine using the far call model in the {**$W-**} state is the same as that of a routine using the near call model, except that a far-return instruction (RETF) is used to return from the routine.

The entry and exit code for a routine using the far call model in the {**$W+**} state (the default) is:

```
inc     BP              ;Indicate FAR frame
push    BP              ;Save odd BP
mov     BP,SP           ;Set up stack frame
push    DS              ;Save DS
sub     SP,LocalSize    ;Allocate locals (if any)
.
.
mov     SP,BP           ;Remove locals and saved DS
pop     BP              ;Restore odd BP
dec     BP              ;Adjust BP
retf    ParamSize       ;Remove parameters and return
```

The entry and exit code for an exportable routine (a procedure or function compiled with the **export** compiler directive) is:

```
mov     AX,DS           ;Load DS selector into AX
nop                     ;Additional space for patching
inc     BP              ;Indicate FAR frame
push    BP              ;Save odd BP
mov     BP,SP           ;Set up stack frame
push    DS              ;Save DS
mov     DS,AX           ;Initialize DS
sub     SP,LocalSize    ;Allocate locals (if any)
push    SI              ;Save SI
push    DI              ;Save DI
.
.
pop     DI              ;Restore DI
pop     SI              ;Restore SI
lea     SP,[BP-2]       ;Deallocate locals (if any)
pop     DS              ;Restore DS
pop     BP              ;Restore odd BP
dec     BP              ;Adjust BP
retf    ParamSize       ;Remove parameters and return
```

For all call models, the instructions required to allocate and deallocate local variables are omitted if the routine has no local variables.

In order to distinguish near and far stack frames, Windows requires that all far stack frames (including stack frames of exported routines) have an odd saved BP value stored in the word at [BP+0]. In addition, Windows requires that the word at [BP-2] contains the data segment selector of the routine's caller. This explains the INC BP, PUSH DS, and DEC BP instructions in the entry and exit code of **far** and **export** routines.

An exported routine must preserve the SI and DI registers, so Turbo Pascal includes instructions in the entry and exit code that

push and pop these registers. For exported routines, Windows requires that the first three bytes of the routine contain the instruction sequence MOV AX,DS followed by a NOP. If the exported routine exists in an application (a callback routine), Windows changes the first three bytes into three NOP instructions, to prepare the routine for use with the Windows *MakeProcInstance* function. If the exported routine exists in a library, Windows changes the first three bytes into a MOV AX,xxxx instruction, where *xxxx* is the selector (segment address) of the library's automatic data segment.

Register-saving conventions

Procedures and functions should preserve the BP, SP, SS, and DS registers. All other registers may be modified. In addition, exported routines should preserve the SI and DI registers.

Exit procedures

By installing an exit procedure, you can gain control over a program's termination process. This is useful when you want to make sure specific actions are carried out before a program terminates; a typical example is updating and closing files.

The *ExitProc* pointer variable allows you to install an exit procedure. The exit procedure always gets called as a part of a program's termination, whether it is a normal termination, a termination through a call to *Halt*, or a termination due to a run-time error.

An exit procedure takes no parameters and must be compiled with a **far** procedure directive to force it to use the far call model.

When implemented properly, an exit procedure actually becomes part of a chain of exit procedures. This chain makes it possible for units as well as programs to install exit procedures. Some units install an exit procedure as part of their initialization code, and then rely on that specific procedure to be called to clean up after the unit; for instance, to close files or to restore interrupt vectors. The procedures on the exit chain get executed in reverse order of installation. This ensures that the exit code of one unit does not get executed before the exit code of any units that depend upon it.

To keep the exit chain intact, you must save the current contents of *ExitProc* before changing it to the address of your own exit procedure. Furthermore, the first statement in your exit procedure must reinstall the saved value of *ExitProc*. The following program demonstrates a skeleton method of implementing an exit procedure:

```
program Testexit;
var
  ExitSave: Pointer;

procedure MyExit; far;
begin
  ExitProc := ExitSave;          { Always restore old vector first }
  ...

end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  ...

end.
```

On entry, the program saves the contents of *ExitProc* in *ExitSave*, and then installs the *MyExit* exit procedure. After having been called as part of the termination process, the first thing *MyExit* does is reinstall the previous exit procedure.

The termination routine in the run-time library keeps calling exit procedures until *ExitProc* becomes **nil**. To avoid infinite loops, *ExitProc* is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to *ExitProc*. If an error occurs in an exit procedure, it will not be called again.

An exit procedure may learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable.

In case of normal termination, *ExitCode* is zero and *ErrorAddr* is **nil**. In case of termination through a call to *Halt*, *ExitCode* contains the value passed to *Halt* and *ErrorAddr* is **nil**. Finally, in case of termination due to a run-time error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the statement in error.

The last exit procedure (the one installed by the run-time library) closes the *Input* and *Output* files. If *ErrorAddr* is not **nil**, it outputs a run-time error message.

If you wish to present run-time error messages yourself, install an exit procedure that examines *ErrorAddr* and outputs a message if it is not **nil**. In addition, before returning, make sure to set *ErrorAddr* to **nil**, so that the error is not reported again by other exit procedures.

Once the run-time library has called all exit procedures, it returns to Windows, passing as a return code the value stored in *ExitCode*.

Interrupt handling

The Turbo Pascal run-time library and the code generated by the compiler are fully interruptible. Also, most of the run-time library is reentrant, which allows you to write interrupt service routines in Turbo Pascal.

Writing interrupt procedures

Interrupt procedures are declared with the **interrupt** directive. Every interrupt procedure must specify the following procedure header (or a subset of it, as explained later):

```
procedure IntHandler(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES,
  BP: Word);
interrupt;
begin
  ...
end;
```

As you can see, all the registers are passed as pseudo-parameters so you can use and modify them in your code. You can omit some or all of the parameters, starting with *Flags* and moving towards *BP*. It is an error to declare more parameters than are listed in the preceding example or to omit a specific parameter without also omitting the ones before it (although no error is reported). For example,

```
procedure IntHandler(DI, ES, BP: Word);              { Invalid call }
procedure IntHandler(SI, DI, DS, ES, BP: Word);        { Valid call }
```

On entry, an interrupt procedure automatically saves all registers (regardless of the procedure header) and initializes the DS register:

```
push    ax
push    bx
```

```
push    cx
push    dx
push    si
push    di
push    ds
push    es
push    bp
mov     bp,sp
sub     sp,LocalSize
mov     ax,SEG DATA
mov     ds,ax
```

Notice the lack of a STI instruction to enable further interrupts. You should code this yourself (if required) using an inline statement. The exit code restores the registers and executes an interrupt-return instruction:

```
mov     sp, bp
pop     bp
pop     es
pop     ds
pop     di
pop     si
pop     dx
pop     cx
pop     bx
pop     ax
iret
```

An interrupt procedure can modify its parameters. Changing the declared parameters will modify the corresponding register when the interrupt handler returns. This can be useful when you are using an interrupt handler as a user service, much like the DOS INT 21H services.

Interrupt procedures that handle hardware-generated interrupts should refrain from using any of Turbo Pascal's input and output or dynamic memory allocation routines, because they are not reentrant. Likewise, no DOS functions can be used because DOS is not reentrant.

C H A P T E R

19

Input and output issues

Text file device drivers

As mentioned in Chapter 11, "The System unit," Turbo Pascal allows you to define your own text file device drivers. A *text file device driver* is a set of four functions that completely implement an interface between Turbo Pascal's file system and some device.

The four functions that define each device driver are *Open*, *InOut*, *Flush*, and *Close*. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where *TTextRec* is the text file record type defined in the earlier section, "File types," in Chapter 3. Each function must be compiled in the {**$F+**} state to force it to use the far call model. The return value of a device interface function becomes the value returned by *IOResult*. The return value of 0 indicates a successful operation.

To associate the device interface functions with a specific file, you must write a customized *Assign* procedure (like the *AssignCrt* procedure in the *WinCrt* unit). The *Assign* procedure must assign the addresses of the four device interface functions to the four function pointers in the text file variable. In addition, it should store the *fmClosed* "magic" constant in the *Mode* field, store the size of the text file buffer in *BufSize*, store a pointer to the text file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device interface functions can use the *UserData* field in the file record to store private information. This field is not modified by the Turbo Pascal file system at any time.

The Open function

The *Open* function is called by the *Reset*, *Rewrite*, and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput*, *fmOutput*, or *fmInOut* to indicate whether the *Open* function was called from *Reset*, *Rewrite*, or *Append*.

The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutput* before *Open* returns.

Open is always called before any of the other device interface functions. For that reason, *Assign* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode*, *Open* can then install pointers to either input- or output-oriented functions. This saves the *InOut*, *Flush*, and *Close* functions from determining the current mode.

The InOut function

The *InOut* function is called by the *Read, Readln, Write, Writeln, Eof, Eoln, SeekEof, SeekEoln,* and *Close* standard procedures and functions whenever input or output from the device is required.

When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into *BufPtr^*, and returns the number of characters read in *BufEnd*. In addition, it stores 0 in *BufPos*. If the *InOut* function returns 0 in *BufEnd* as a result of an input request, *Eof* becomes True for the file.

When *Mode* is *fmOutput*, the *InOut* function writes *BufPos* characters from *BufPtr^*, and returns 0 in *BufPos*.

The Flush function

The *Flush* function is called at the end of each *Read, Readln, Write,* and *Writeln*. It can optionally flush the text file buffer.

If *Mode* is *fmInput*, the *Flush* function can store 0 in *BufPos* and *BufEnd* to flush the remaining (un-read) characters in the buffer. This feature is seldom used.

If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer, exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text will not appear on the device until the buffer becomes full or the file is closed.

The Close function

The *Close* function is called by the *Close* standard procedure to close a text file associated with a device. (The *Reset, Rewrite,* and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutput*, then before calling *Close*, Turbo Pascal's file system calls *InOut* to ensure that all characters have been written to the device.

Direct port access

For access to the 80x86 CPU data ports, Turbo Pascal implements two predefined arrays, *Port* and *PortW*. Both are one-dimensional

arrays, and each element represents a data port, whose port address corresponds to its index. The index type is the integer type Word. Components of the *Port* array are of type byte, and components of the *PortW* array are of type Word.

When a value is assigned to a component of *Port* or *PortW*, the value is output to the selected port. When a component of *Port* or *PortW* is referenced in an expression, its value is input from the selected port. Some examples include:

```
Port[$20] := $20;
Port[Base] := Port[Base] xor Mask;
while Port[$B2] and $80 = 0 do  { Wait };
```

Use of the *Port* and *PortW* arrays is restricted to assignment and reference in expressions only, that is, components of *Port* and *PortW* cannot be used as variable parameters. Furthermore, references to the entire *Port* or *PortW* array (reference without index) are not allowed.

C H A P T E R

20

Automatic optimizations

Turbo Pascal performs several different types of code optimizations, ranging from constant folding and short-circuit Boolean expression evaluation all the way up to smart linking. The following sections describe some of the types of optimizations performed.

Constant folding

If the operand(s) of an operator are constants, Turbo Pascal evaluates the expression at compile time. For example,

```
X := 3 + 4 * 2
```

generates the same code as `X := 11`, and

```
S := 'In' + 'Out'
```

generates the same code as `S := 'InOut'`.

Likewise, if an operand of an *Abs*, *Chr*, *Hi*, *Length*, *Lo*, *Odd*, *Ord*, *Pred*, *Ptr*, *Round*, *Succ*, *Swap*, or *Trunc* function call is a constant, the function is evaluated at compile time.

If an array index expression is a constant, the address of the component is evaluated at compile time. For example, accessing *Data*[5, 5] is just as efficient as accessing a simple variable.

Constant merging

Using the same string constant two or more times in a statement part generates only one copy of the constant. For example, two or more `Write('Done')` statements in the same statement part will reference the same copy of the string constant 'Done'.

Short-circuit evaluation

Turbo Pascal implements short-circuit Boolean evaluation, which means that evaluation of a Boolean expression stops as soon as the result of the entire expression becomes evident. This guarantees minimum execution time, and usually minimum code size. Short-circuit evaluation also makes possible the evaluation of constructs that would not otherwise be legal; for instance:

```
while (I <= Length(S)) and (S[I] <> ' ') do
  Inc(I);
while (P <> nil) and (P^.Value <> 5) do
  P := P^.Next;
```

In both cases, the second test is not evaluated if the first test is False.

The opposite of short-circuit evaluation is complete evaluation, which is selected through a {**$B+**} compiler directive. In this state, every operand of a Boolean expression is guaranteed to be evaluated.

Order of evaluation

As permitted by the Pascal standards, operands of an expression are frequently evaluated differently from the left to right order in which they are written. For example, the statement

```
I := F(J) div G(J);
```

where *F* and *G* are functions of type Integer, causes *G* to be evaluated before *F*, since this enables the compiler to produce better code. Because of this, it is important that an expression never depend on any specific order of evaluation of the

embedded functions. Referring to the previous example, if *F* must be called before *G*, use a temporary variable:

```
T := F(J);
I := T div G(J);
```

As an exception to this rule, when short-circuit evaluation is enabled (the {**$B-**} state), Boolean operands grouped with **and** or **or** are *always* evaluated from left to right.

Range checking

Assignment of a constant to a variable and use of a constant as a value parameter is range-checked at compile time; no run-time range-check code is generated. For example, `X := 999`, where *X* is of type Byte, causes a compile-time error.

Shift instead of multiply

The operation *X* * *C*, where *C* is a constant and a power of 2, is coded using a **SHL** instruction.

Likewise, when the size of an array's components is a power of 2, a **SHL** instruction (not a **MUL** instruction) is used to scale the index expression.

Automatic word alignment

By default, Turbo Pascal aligns all variables and typed constants larger than 1 byte on a machine-word boundary. On all 16-bit 80x86 CPUs, word alignment means faster execution, since word-sized items on even addresses are accessed faster than words on odd addresses.

For further details, refer to Chapter 21, "Compiler directives."

Data alignment is controlled through the **$A** compiler directive. In the default {**$A+**} state, variables and typed constants are aligned as described above. In the {**$A-**} state, no alignment measures are taken.

Dead code removal

Statements that are known never to execute do not generate any code. For example, these constructs don't generate any code:

```
if False then
  statement
while False do
  statement
```

Smart linking

When compiling to memory, Turbo Pascal's smart linker is disabled. This explains why some programs become smaller when compiled to disk.

Turbo Pascal's built-in linker automatically removes unused code and data when building an .EXE file. Procedures, functions, variables, and typed constants that are part of the compilation, but never get referenced, are removed from the .EXE file. The removal of unused code takes place on a per procedure basis; the removal of unused data takes place on a per declaration section basis.

Consider the following program:

```
program SmartLink;

const
  H: array[0..15] of Char = '0123456789ABCDEF';

var
  I, J: Integer;
  X, Y: Real;

var
  S: string[79];

var
  A: array[1..10000] of Integer;

procedure P1;
begin
  A[1] := 1;
end;

procedure P2;
begin
  I := 1;
end;
```

```
procedure P3;
begin
  S := 'Turbo Pascal';
  P2;
end;

begin
  P3;
end.
```

The main program calls *P3*, which calls *P2*, so both *P2* and *P3* are included in the .EXE file; and since *P2* references the first **var** declaration section, and *P3* references the second **var** declaration, *I*, *J*, *X*, *Y*, and *S* are also included in the .EXE file. However, no references are made to *P1*, and none of the included procedures reference *H* and *A*, so these objects are removed.

Smart linking is especially valuable in connection with units that implement procedure/function libraries. An example of such a unit is the *WinDos* standard unit: It contains a number of procedures and functions, all of which are seldom used by the same program. If a program uses only one or two procedures from *WinDos*, then only these procedures are included in the final .EXE file, and the remaining ones are removed, greatly reducing the size of the .EXE file.

C H A P T E R

21

Compiler directives

Some of the Turbo Pascal compiler's features are controlled through *compiler directives*. A compiler directive is a comment with a special syntax. Turbo Pascal allows compiler directives wherever comments are allowed.

A compiler directive starts with a **$** as the first character after the opening comment delimiter, and is immediately followed by a name (one or more letters) that designates the particular directive. There are three types of directives:

- **Switch directives.** These directives turn particular compiler features on or off by specifying + or – immediately after the directive name.
- **Parameter directives.** These directives specify parameters that affect the compilation, such as file names and memory sizes.
- **Conditional directives.** These directives control conditional compilation of parts of the source text, based on user-definable conditional symbols.

All directives, except switch directives, must have at least one blank between the directive name and the parameters. Here are some examples of compiler directives:

```
{$B+}
{$R- Turn off range checking}
{$I TYPES.INC}
{$O EdFormat}
{$M 65520,8192,655360}
{$DEFINE Debug}
```

```
{$IFDEF Debug}
{$ENDIF}
```

You can put compiler directives directly into your source code. You can also change the default directives for both the command-line compiler (TPWC.EXE) and the IDE (TPW.EXE). The Options | Compiler menu contains many of the compiler directives; any changes you make to the settings there will affect all subsequent compilations. When using the command-line compiler, you can specify compiler directives on the command line (for example, `TPWC /$R+ MYPROG`), or you can place directives in a configuration file (see Chapter 8 of the *User's Guide* for information). Compiler directives in the source code always override the default values in both the command-line compiler and the IDE.

If you are working in the IDE, using the editor's Alternate command set, and want a quick way to see what compiler directives are in effect, press *Ctrl+O O*. Turbo Pascal will insert the current settings at the top of your edit window.

Switch directives

Switch directives are either *global* or *local*. Global directives affect the entire compilation, whereas local directives affect only the part of the compilation that extends from the directive until the next occurrence of the same directive.

Global directives must appear before the declaration part of the program or the unit being compiled, that is, before the first **uses**, **label**, **const**, **type**, **procedure**, **function**, or **begin** keyword. Local directives, on the other hand, can appear anywhere in the program or unit.

Multiple switch directives can be grouped in a single compiler directive comment by separating them with commas; for example,

```
{$B+,R-,S-}
```

There can be no spaces between the directives in this case.

Align data

Syntax {$A+} or {$A-}

Default {$A+}

Type Global

Menu equivalent Options | Compiler | Word Align Data

Remarks The **$A** directive switches between byte and word alignment of variables and typed constants. Word alignment has no effect on the 8088 CPU. However, on all 80x86 CPUs, word alignment means faster execution, since word-sized items on even addresses are accessed in one memory cycle, in comparison to two memory cycles for words on odd addresses.

In the {**$A+**} state, all variables and typed constants larger than one byte are aligned on a machine-word boundary (an even-numbered address). If required, unused bytes are inserted between variables to achieve word alignment. The {**$A+**} directive does not affect byte-sized variables; neither does it affect fields of record structures and elements of arrays. A field in a record will align on word boundary only if the total size of all fields before it is even. Likewise, for every element of an array to align on a word boundary, the size of the elements must be even.

In the {**$A-**} state, no alignment measures are taken. Variables and typed constants are simply placed at the next available address, regardless of their size.

Regardless of the state of the **$A** directive, each global **var** and **const** declaration section always starts at a word boundary. Likewise, the compiler always keeps the stack pointer (SP) word aligned, by allocating an extra unused byte in a procedure's stack frame if required.

Boolean evaluation

Syntax {$B+} or {$B-}

Default {$B-}

Type Local

Menu equivalent Options | Compiler | Boolean Evaluation

Remarks The **$B** directive switches between the two different models of code generation for the **and** and **or** Boolean operators.

In the {**$B+**} state, the compiler generates code for complete Boolean expression evaluation. This means that every operand of a Boolean expression, built from the **and** and **or** operators, is guaranteed to be evaluated, even when the result of the entire expression is already known.

In the {**$B-**} state, the compiler generates code for short-circuit Boolean expression evaluation, which means that evaluation stops as soon as the result of the entire expression becomes evident.

For further details, refer to the section "Boolean operators" in Chapter 6, "Expressions."

Debug information

Syntax `{$D+}` or `{$D-}`

Default `{$D+}`

Type Global

Menu equivalent Options | Compiler | Debug Information

Remarks The **$D** directive enables or disables the generation of debug information. This information consists of a line-number table for each procedure, which maps object code addresses into source text line numbers.

For units, the debug information is recorded in the .TPU file along with the unit's object code. Debug information increases the size of .TPU files, and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

$D will not take effect unless you have checked the Debug Info in EXE checkbox (Options | Linker) in the IDE or specified **/V** on the command line when using TPWC.EXE.

When the Debug Information option is checked for a given program or unit, you can use Turbo Debugger for Windows to single-step and set breakpoints in that module.

The Map File (Options | Linker) options produce complete line information for a given module only if you've compiled that module in the {**$D+**} state.

The **$D** switch is usually used in conjunction with the **$L** switch, which enables and disables the generation of local symbol information for debugging.

Force far calls

Syntax {$F+} or {$F-}

Default {$F-}

Type Local

Menu equivalent Options | Compiler | Force Far Calls

Remarks The **$F** directive controls which call model to use for subsequently compiled procedures and functions. Procedures and functions compiled in the {**$F+**} state always use the far call model. In the {**$F-**} state, Turbo Pascal automatically selects the appropriate model: far if the procedure or function is declared in the **interface** section of a unit; near otherwise.

The near and far call models are described in full in Chapter 18, "Control issues."

Generate 80286 Code

Syntax {$G+} or {$G-}

Default {$G-}

Type Local

Menu equivalent Options | Compiler | 286 code

The **$G** directive enables or disables 80286 code generation. In the {**$G-**} state, only generic 8086 instructions are generated, and programs compiled in this state can run on any 80x86 family processor. In the {**$G+**} state, the compiler uses the additional instructions of the 80286 to improve code generation, but programs compiled in this state cannot run on 8088 and 8086 processors. Additional instructions used in the {**$G+**} state include **ENTER**, **LEAVE**, **PUSH** immediate, extended **IMUL**, and extended **SHL** and **SHR**.

Input/output checking

Syntax {$I+} or {$I-}

Default {$I+}

Type Local

Menu equivalent Options | Compiler | I/O Checking

Remarks The **$I** directive enables or disables the automatic code generation that checks the result of a call to an I/O procedure. I/O procedures are described in Chapter 19, "Input and output issues." If an I/O procedure returns a nonzero I/O result when this switch is on, the program terminates, displaying a run-time error message. When this switch is off, you must check for I/O errors by using the *IOResult* function.

Local symbol information

Syntax {$L+} or {$L-}

Default {$L+}

Type Global

Menu equivalent Options | Compiler | Local Symbols

Remarks The **$L** directive enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module, that is, the symbols in the module's implementation part, and the symbols within the module's procedures and functions.

For units, the local symbol information is recorded in the .TPU file along with the unit's object code. Local symbol information increases the size of .TPU files, and takes up additional room when compiling programs that use the unit, but it does not affect the size or speed of the executable program.

$L will not take effect unless you have checked the Debug Info in EXE checkbox (Options | Linker) in the IDE or specified **/V** on the command line when using TPCW.EXE.

When local symbols are on for a given program or unit, you can use Turbo Debugger for Windows to examine and modify the module's local variables using the symbol names used in your module. This will make

debugging much easier. Furthermore, calls to the module's procedures and functions can be examined via the View | Stack window.

The Map File (Options | Linker) and Debug Info (Options | Linker) options produce local symbol information for a given module only if that module was compiled in the {**$L+**} state.

The **$L** switch is usually used in conjunction with the **$D** switch, which enables and disables the generation of line-number tables for debugging. Note that the **$L** directive is ignored if the compiler is in the {**$D-**} state.

Range checking

Syntax {$R+} or {$R-}

Default {$R-}

Type Local

Menu equivalent Options | Compiler | Range Checking

Remarks The **$R** directive enables or disables the generation of range-checking code. In the {**$R+**} state, all array and string-indexing expressions are verified as being within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. If a range check fails, the program terminates and displays a run-time error message.

If **$R** is switched on, all calls to virtual methods are checked for the initialization status of the object instance making the call. If the instance making the call has not been initialized by its constructor, a range check run-time error occurs.

Enabling range checking and virtual method call checking slows down your program and makes it somewhat larger, so use the {**$R+**} only for debugging.

Stack-overflow checking

Syntax {$S+} or {$S-}

Default {$S+}

Type Local

Menu equivalent Options | Compiler | Stack Checking

Remarks The **$S** directive enables or disables the generation of stack-overflow checking code. In the {**$S+**} state, the compiler generates code at the beginning of each procedure or function that checks whether there is sufficient stack space for the local variables and other temporary storage. When there is not enough stack space, a call to a procedure or function compiled with {**$S+**} causes the program to terminate and display a run-time error message. In the {**$S-**} state, such a call is most likely to cause a system crash.

Var-string checking

Syntax {$V+} or {$V-}

Default {$V+}

Type Local

Menu equivalent Options | Compiler | String Var Checking

Remarks The **$V** directive controls type checking on strings passed as variable parameters. In the {**$V+**} state, strict type checking is performed, requiring the formal and actual parameters to be of *identical* string types. In the {**$V-**} (relaxed) state, any string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

Windows stack frames

Syntax {$W+} or {$W-}

Default {$W+}

Type Local

Menu equivalent Options | Compiler | Windows Stack Frame

Remarks The **$W** directive controls the generation of Windows specific procedure entry and exit code for **far** procedures and functions. In the {**$W+**} state, special entry and exit code is generated for **far** procedures and functions so that the Windows real mode memory manager can correctly identify **far** stack frames when making adjustments to the call chain after moving a code or data segment. In the {**$W-**} state, this additional entry and exit

code is not generated. **$W** can be disabled if a program is to run in Windows standard mode or Windows 386 enhanced mode only.

Extended syntax

Syntax `{$X+}` or `{$X-}`

Default `{$X+}`

Type Global

Menu equivalent Options | Compiler | Extended Syntax

The **$X** directive enables or disables Turbo Pascal's extended syntax:

- Function statements. In the **{$X+}** mode, functions calls can be used as statements, that is the result of a function call can be discarded. Generally, the computations performed by a function are represented through its result, so discarding the result makes little sense. However, in certain cases a function can carry out multiple operations based on its parameters, and some of those cases may not produce a sensible result—in such cases, the **{$X+}** extensions allow the function to be treated as a procedure.

  Note: The **{$X+}** directive does not apply to built-in functions (that is, functions defined in the *System* unit).
- Null-terminated strings. A **{$X+}** compiler directive enables Turbo Pascal's support for null-terminated strings by activating the special rules that apply to the built-in *PChar* type and zero-based character arrays. For more details about null-terminated strings, refer to Chapter 13, "The Strings unit."

Parameter directives

Parameter directives permit you to specify parameters which affect how a program is compiled.

Code segment attribute

Syntax `{$C attribute attribute}`

Default {$C MOVEABLE PRELOAD PERMANENT}

Type Global

Remarks The **$C** directive is used to control the attributes of a code segment. Every code segment in an application or library has a set of attributes that determine the behavior of the code segment when it is loaded into memory. For example, you can specify that a code segment is *moveable* meaning that Windows can move the code segment around in memory as needed, or you can specify that a code segment is *fixed* meaning that the location of the code segment in memory cannot change.

A **$C** directive affects only the code segment of the module (unit, program, or library) in which it is placed. In the following table, the code segment attribute options occur in groups of two; each option has an opposite toggle. Here are the grouped options:

MoveAble	Windows can change the location of the code segment in memory.
Fixed	Windows can't change the location of the code segment in memory.
Preload	The code segment loads when your program begins execution.
DemandLoad	The code segment loads only when it is needed.
Permanent	Once Windows loads the code segment, it remains in memory.
Discardable	The code segment can be unloaded when it is no longer needed.

The first option of each group is the default. You may specify multiple code segment attributes using the **$C** directive. If both options of a group in a **$C** directive are specified, only the last one will take effect. For example,

```
{$C Fixed Moveable Discardable }
```

will make the code segment moveable and it can be discarded when it is no longer needed.

Description

Syntax `{$D text}`

Type Global

Remarks The **$D** directive inserts the text you specify into an EXE file or DLL. Traditionally the text is a copyright notice or a version number, but you may specify any text of your choosing. Here is an example:

```
{$D My Masterpiece version 12.5 (C) Fly By Nite Software, 1991}
```

Include file

Syntax `{$I filename}`

Type Local

Remarks The **$I** directive instructs the compiler to include the named file in the compilation. In effect, the file is inserted in the compiled text right after the {**$I** *filename*} directive. The default extension for *filename* is .PAS. If *filename* does not specify a directory, then, in addition to searching for the file in the current directory, Turbo Pascal searches in the directories specified in the **O**ptions | **D**irectories | **I**nclude Directories input box (or in the directories specified in the **/I** option on the TPC command line).

You can nest Include files up to 15 levels deep.

There is one restriction to the use of Include files: An Include file cannot be specified in the middle of a statement part. In fact, all statements between the **begin** and **end** of a statement part must reside in the same source file.

Link object file

Syntax `{$L filename}`

Type Local

Remarks The Link object file directive instructs the compiler to link the named file with the program or unit being compiled. The **$L** directive is used to link with code written in assembly language for subprograms declared to be **external**. The named file must be an Intel relocatable object file (.OBJ file). The default extension for *filename* is .OBJ. If *filename* does not specify a

directory, then, in addition to searching for the file in the current directory, Turbo Pascal searches in the directories specified in the Options | Directories | Object Directories input box (or in the directories specified in the **/O** option on the TPWC command line). For further details about linking with assembly language, see Chapter 23, "Linking assembler code."

Memory allocation sizes

Syntax `{$M stacksize,heapsize}`

Default `{$M 8192,8192}`

Type Global

Menu equivalent Options | Memory Sizes

Remarks The **$M** directive specifies an application or library's memory allocation parameters. *stacksize* must be an integer number in the range 1024 to 65520, which specifies the size of the stack area in an application's data segment. *stacksize* is ignored in a library (a library uses the stack of applications that call it). *heapsize* must be an integer number in the range 0 to 65520, which specifies the size of the local heap area in the data segment.

The stack segment and the heap are further discussed in Chapter 4, "Variables," and Chapter 16, "Memory issues."

The **$M** directive has no effect when used in a unit.

Numeric coprocessor

Syntax `{$N+}` or `{$N-}`

Default `{$N-}`

Type Global

Menu equivalent Options | Compiler | 80x87 code

Remarks The **$N** directive switches between the two different models of floating-point code generation supported by Turbo Pascal. In the **{$N-}** state, code is generated to perform all real-type calculations in software by calling run-time library routines. In the **{$N+}** state, code is generated to perform all Real-type calculations using the 80x87 numeric coprocessor.

Resource file

Syntax `{$R Filename}`

Type Local

Remarks The **$R** directive specifies the name of a resource file to be included in an application or library. The named file must be a Windows resource file, and the default extension for *filename* is .RES.

When a {**$R** filename} directive is used in a unit, the specified filename is simply recorded in the resulting .TPU file. No checks are made at that point to ensure that the filename is correct and that it specifies an existing file.

When an application or library is linked (after compiling the program or library source file), the resource files specified in all used units as well as in the program or library itself are processed, and each resource in each resource file is copied to the .EXE or .DLL being produced. During the resource processing phase, Turbo Pascal's linker searches for .RES files in the current directory and in the directories specified in the O | D | Resource Directories input box (or in the directories specified using a **/R** directive on the TPCW command line).

Conditional compilation

Turbo Pascal's conditional compilation directives allow you to produce different code from the same source text, based on conditional symbols.

There are two basic conditional compilation constructs, which closely resemble Pascal's **if** statement. The first construct

```
{$IFxxx} ... {$ENDIF}
```

causes the source text between {**$IF***xxx*} and {**$ENDIF**} to be compiled only if the condition specified in {**$IF***xxx*} is True; if the condition is False, the source text between the two directives is ignored.

The second conditional compilation construct

```
{$IFxxx} ... {$ELSE} ... {$ENDIF}
```

causes either the source text between {**$IF***xxx*} and {**$ELSE**} or the source text between {**$ELSE**} and {**$ENDIF**} to be compiled, based on the condition specified by the {**$IF***xxx*}.

Here are some examples of conditional compilation constructs:

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}

{$IFDEF CPU87}
  {$N+}
  type
    Real = Double;
{$ELSE}
  {$N-}
  type
    Single = Real;
    Double = Real;
    Extended = Real;
    Comp = Real;
{$ENDIF}
```

You can nest conditional compilation constructs up 16 levels deep. For every {**$IF***xxx*}, the corresponding {**$ENDIF**} must be found within the same source file—which means there must be an equal number of {**$IF***xxx*}'s and {**$ENDIF**}'s in every source file.

Conditional symbols

Conditional compilation is based on the evaluation of conditional symbols. Conditional symbols are defined and undefined (forgotten) using the directives

```
{$DEFINE name}
{$UNDEF name}
```

You can also use the **/D** switch in the command-line compiler (or place it in the Conditional Defines input box from within **O**ptions | **C**ompiler of the IDE).

Conditional symbols are best compared to Boolean variables: They are either True (defined) or False (undefined). The {**$DEFINE**} directive sets a given symbol to True, and the {**$UNDEF**} directive sets it to False.

Conditional symbols follow the exact same rules as Pascal identifiers: They must start with a letter, followed by any combination

of letters, digits, and underscores. They can be of any length, but only the first 63 characters are significant.

Important! Conditional symbols and Pascal identifiers have no correlation whatsoever. Conditional symbols cannot be referenced in the actual program, and the program's identifiers cannot be referenced in conditional directives. For example, the construct

```
const
  Debug = True;
begin
  {$IFDEF Debug}
   Writeln('Debug is on');
  {$ENDIF}
end;
```

will *not* compile the *Writeln* statement. Likewise, the construct

```
{$DEFINE Debug}
begin
  if Debug then
   Writeln('Debug is on');
end;
```

will result in an unknown identifier error in the **if** statement.

Turbo Pascal defines the following standard conditional symbols:

VER10 Always defined, indicating that this is version 1.0 of Turbo Pascal for Windows. Future versions will have corresponding predefined symbols; for example, version 2.0 would have *VER20* defined, version 2.5 would have *VER25* defined, and so on.

WINDOWS Always defined, indicating that the operating environment is MS-Windows. Versions of Turbo Pascal for other operating environments or operating systems will instead define a symbolic name for that particular operating environment or operating system.

CPU86 Always defined, indicating that the CPU belongs to the 80x86 family of processors. Versions of Turbo Pascal for other CPUs will instead define a symbolic name for that particular CPU.

CPU87 Defined if an 80x87 numeric coprocessor is present at compile time. If the construct

```
{$IFDEF CPU87} {$N+} {$ELSE} {$N-} {$ENDIF}
```

appears at the beginning of a compilation, Turbo Pascal automatically selects the appropriate model of floating-point code generation for that particular computer.

Other conditional symbols can be defined before a compilation by using the Conditional Defines input box (Options | Compiler), or the **/D** command-line option if you are using TPCW.

The DEFINE directive

Syntax `{$DEFINE name}`

Remarks Defines a conditional symbol of *name*. The symbol is recognized for the remainder of the compilation of the current module in which the symbol is declared, or until it appears in an {**$UNDEF** *name*} directive. The {**$DEFINE** *name*} directive has no effect if *name* is already defined.

The UNDEF directive

Syntax `{$UNDEF name}`

Remarks Undefines a previously defined conditional symbol. The symbol is forgotten for the remainder of the compilation or until it reappears in a {**$DEFINE** *name*} directive. The {**$UNDEF** *name*} directive has no effect if *name* is already undefined.

The IFDEF directive

Syntax `{$IFDEF name}`

Remarks Compiles the source text that follows it if *name* is defined.

The IFNDEF directive

Syntax `{$IFNDEF name}`

Remarks Compiles the source text that follows it if *name* is not defined.

The IFOPT directive

Syntax `{$IFOPT switch}`

Remarks Compiles the source text that follows it if *switch* is currently in the specified state. *switch* consists of the name of a switch option, followed by a **+** or a **–** symbol. For example, the construct

```
{$IFOPT N+}
  type Real = Extended;
{$ENDIF}
```

will compile the type declaration if the **$N** option is currently active.

The ELSE directive

Syntax `{$ELSE}`

Remarks Switches between compiling and ignoring the source text delimited by the last {**$IF***xxx*} and the next {**$ENDIF**}.

The ENDIF directive

Syntax `{$ENDIF}`

Remarks Ends the conditional compilation initiated by the last {**$IF***xxx*} directive.

PART 4

Using Turbo Pascal with assembly language

C H A P T E R

22

The inline assembler

Turbo Pascal's inline assembler allows you to write 8086/8087 and 80286/80287 assembler code directly inside your Pascal programs. Of course, you can still convert assembler instructions to machine code manually for use in **inline** statements, or link in .OBJ files that contain **external** procedures and functions when you want to mix Pascal and assembler.

The inline assembler implements a large subset of the syntax supported by Turbo Assembler and Microsoft's Macro Assembler. The inline assembler supports all 8086/8087 and 80286/80287 opcodes, and all but a few of Turbo Assembler's expression operators.

Except for **DB**, **DW**, and **DD** (define byte, word, and double word), none of Turbo Assembler's directives, such as **EQU**, **PROC**, **STRUC**, **SEGMENT**, and **MACRO**, are supported by the inline assembler. Operations implemented through Turbo Assembler directives, however, are largely matched by corresponding Turbo Pascal constructs. For example, most **EQU** directives correspond to **const**, **var**, and **type** declarations in Turbo Pascal, the **PROC** directive corresponds to **procedure** and **function** declarations, and the **STRUC** directive corresponds to Turbo Pascal **record** types. In fact, Turbo Pascal's inline assembler can be thought of as an assembler language compiler that uses Pascal syntax for all declarations.

The **asm** statement

The inline assembler is accessed through **asm** statements. The syntax of an **asm** statement is

```
asm AsmStatement < Separator AsmStatement > end
```

where *AsmStatement* is an assembler statement, and *Separator* is a semicolon, a new-line, or a Pascal comment. Here are some examples of **asm** statements:

```
if EnableInts then
  asm
    sti
  end
  else
  asm
    cli
  end;

asm
  mov ax,Left; xchg ax,Right; mov Left,ax;
end;

asm
  mov     ah,0                          { Read keyboard function code }
  int     16H                             { Call PC BIOS to read key }
  mov     CharCode,al                             { Save ASCII code }
  mov     ScanCode,ah                              { Save scan code }
end;

asm
  push    ds                                                 { Save DS }
  lds     si,Source                          { Load source pointer }
  les     di,Dest                          { Load destination pointer }
  mov     cx,Count                                 { Load block size }
  cld                                                { Move forwards }
  rep     movsb                                         { Copy block }
  pop     ds                                            { Restore DS }
end;
```

Notice that multiple assembler statements can be placed on one line if they are separated by semicolons. Also notice that a semicolon is not required between two assembler statements if the statements are on separate lines. Finally, notice that a semicolon does *not* indicate that the rest of the line is a comment—comments must be written in Pascal style using **{** and **}** or **(*** and ***)**.

Register use

The rules of register use in an **asm** statement are in general the same as those of an **external** procedure or function. An **asm** statement must preserve the BP, SP, SS, and DS registers, but can freely modify the AX, BX, CX, DX, SI, DI, ES, and Flags registers. On entry to an **asm** statement, BP points to the current stack frame, SP points to the top of the stack, SS contains the segment address of the stack segment, and DS contains the segment address of the data segment. Except for BP, SP, SS, and DS, an **asm** statement can assume nothing about register contents on entry to the statement.

Assembler statement syntax

The syntax of an assembler statement is

```
[ Label ":" ] < Prefix > [ Opcode [ Operand < "," Operand > ] ]
```

where *Label* is a label identifier, *Prefix* is an assembler prefix opcode (operation code), *Opcode* is an assembler instruction opcode or directive, and *Operand* is an assembler expression.

Comments are allowed between assembler statements, but not within them. For example, this is allowed:

```
asm
  mov ax,1 {Initial value}
  mov cx,100 {Count}
end;
```

but this is an error:

```
asm
  mov {Initial value} ax,1;
  mov cx, {Count} 100
end;
```

Labels

Labels are defined in assembler just as in Pascal, by writing a label identifier and a colon before a statement; and just as in Pascal, labels defined in assembler must be declared in a **label** declaration part in the block containing the **asm** statement. There is however one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). Since an at-sign cannot be part of a Pascal identifier, such local labels are automatically restricted to use within **asm** statements. A local label is known only within the **asm** statement that defines it (that is, the scope of a local label extends from the **asm** keyword to the **end** keyword of the **asm** statement that contains it).

Unlike a normal label, a local label does not have to be declared in a **label** declaration part before it is used.

The exact composition of a local label identifier is an at-sign (@) followed by one or more letters (*A..Z*), digits (0..9), underscores (_), or at-signs. As with all labels, the identifier is followed by a colon (:).

The following program fragment demonstrates use of normal and local labels in **asm** statements:

```
label Start, Stop;

  ...

begin
  asm
    Start:

    ...

      jz        Stop
    @1:
      .
      .
      loop    @1
  end;
  asm
    @1:
      .
      .
      jc      @2
      .
      .
      jmp     @1
    @2:
  end;
  goto Start;
  Stop:
end;
```

Notice that a normal label can be defined within an **asm** statement and referenced outside an **asm** statement and vice

versa. Also, notice that the same local label name can be used in different **asm** statements.

Prefix opcodes

The inline assembler supports the following prefix opcodes:

LOCK	Bus lock
REP	Repeat string operation
REPE/REPZ	Repeat string operation while equal/zero
REPNE/REPNZ	Repeat string operation while not equal/zero
SEGCS	CS (code segment) override
SEGDS	DS (data segment) override
SEGES	ES (extra segment) override
SEGSS	SS (stack segment) override

Zero or more of these can prefix an assembler instruction. For example,

```
asm
  rep movsb              { Move CX bytes from DS:SI to ES:DI }
  SEGES lodsw            { Load word from ES:SI }
  SEGCS mov ax,[bx]      { Same as MOV AX,CS:[BX] }
  SEGES                  { Affects next assembler statement }
  mov WORD PTR [DI],0    { Becomes MOV WORD PTR ES:[DI],0 }
end;
```

Notice that a prefix opcode can be specified without an instruction opcode in the same statement—in that case, the prefix opcode affects the instruction opcode in the following assembler statement.

An instruction opcode seldom, if ever, has more than one prefix opcode, and at most no more than three prefix opcodes can make sense (a **LOCK**, followed by a **SEG*xx***, followed by a **REP*xx***). Be careful about using multiple prefix opcodes—ordering is important, and some 80x86 processors do not handle all combinations correctly. For example, an 8086 or 8088 "remembers" only the **REP*xx*** prefix if an interrupt occurs in the middle of a repeated string instruction, so a **LOCK** or **SEG*xx*** prefix cannot safely be coded before a **REP*xx*** string instruction.

Instruction opcodes

The inline assembler supports all 8086/8087 and 80286/80287 instruction opcodes. 8087 opcodes are available only in the **{$N+}** state (numeric processor enabled), 80286 opcodes are available

only in the {**$G+**} state (80286 code generation enabled), and 80287 opcodes are available only in the {**$G+,N+**} state.

For a complete description of each instruction, refer to your 80x86 and 80x87 reference manuals.

RET instruction sizing

The **RET** instruction opcode generates a near return or a far return machine code instruction depending on the call model of the current procedure or function.

```
procedure NearProc; near;
begin
  asm
    ret    { Generates a near return }
  end;
end;

procedure FarProc; far;
begin
  asm
    ret    { Generates a far return }
  end;
end;
```

The **RETN** and **RETF** instructions on the other hand always generate a near return and a far return, regardless of the call model of the current procedure or function.

Automatic jump sizing

Unless otherwise directed, the inline assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (**JMP**), and all conditional jump instructions, when the target is a label (not a procedure or function).

For an unconditional jump instruction (**JMP**), the inline assembler generates a short jump (one byte opcode followed by a one byte displacement) if the distance to the target label is within –128 to 127 bytes; otherwise a near jump (one byte opcode followed by a two byte displacement) is generated.

For a conditional jump instruction, a short jump (1 byte opcode followed by a 1 byte displacement) is generated if the distance to the target label is within –128 to 127 bytes; otherwise, the inline assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (5 bytes in total). For example, the assembler statement

```
JC      Stop
```

where *Stop* is not within reach of a short jump is converted to a machine code sequence that corresponds to

```
jnc     Skip
jmp     Stop
Skip:
```

Jumps to the entry points of procedures and functions are always either near or far, but never short, and conditional jumps to procedures and functions are not allowed. You can force the inline assembler to generate an unconditional near jump or far jump to a label by using a **NEAR PTR** or **FAR PTR** construct. For example, the assembler statements

```
jmp     NEAR PTR Stop
jmp     FAR PTR Stop
```

will always generate a near jump and a far jump, respectively, even if *Stop* is a label within reach of a short jump.

Assembler directives

Turbo Pascal's inline assembler supports three assembler directives: **DB** (define byte), **DW** (define word), and **DD** (define double word). They each generate data corresponding to the comma-separated operands that follow the directive.

The **DB** directive generates a sequence of bytes. Each operand may be a constant expression with a value between –128 and 255, or a character string of any length. Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The **DW** directive generates a sequence of words. Each operand may be a constant expression with a value between –32,768 and 65,535, or an address expression. For an address expression, the inline assembler generates a near pointer, that is, a word that contains the offset part of the address.

The **DD** directive generates a sequence of double words. Each operand may be a constant expression with a value between –2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the inline assembler generates a far pointer, that is, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The data generated by the **DB**, **DW**, and **DD** directives is always stored in the code segment, just like the code generated by other inline assembler statements. To generate uninitialized or initialized data in the data segment, you should use normal Pascal **var** or **const** declarations.

Some examples of **DB**, **DW**, and **DD** directives follow:

```
asm
  DB    0FFH                                        { One byte }
  DB    0,99                                       { Two bytes }
  DB    'A'                                         { Ord('A') }
  DB    'Hello world...',0DH,0AH          { String followed by CR/LF }
  DB    12,"Turbo Pascal"                    { Pascal style string }
  DW    0FFFFH                                      { One word }
  DW    0,9999                                     { Two words }
  DW    'A'                                  { Same as DB 'A',0 }
  DW    'BA'                                 { Same as DB 'A','B' }
  DW    MyVar                                  { Offset of MyVar }
  DW    MyProc                                 { Offset of MyProc }
  DD    0FFFFFFFFH                              { One double-word }
  DD    0,999999999                            { Two double-words }
  DD    'A'                                { Same as DB 'A',0,0,0 }
  DD    'DCBA'                       { Same as DB 'A','B','C','D' }
  DD    MyVar                                  { Pointer to MyVar }
  DD    MyProc                                 { Pointer to MyProc }
end;
```

In Turbo Assembler, when an identifier precedes a **DB**, **DW**, or **DD** directive, it causes declaration of a byte, word, or double-word sized variable at the location of the directive. For example, Turbo Assembler allows the following:

```
ByteVar     DB      ?
WordVar     DW      ?

...

            mov     al,ByteVar
            mov     bx,WordVar
```

The inline assembler does not support such variable declarations. In Turbo Pascal, the only kind of symbol that can be defined in an inline assembler statement is a label. All variables must be declared using Pascal syntax, and the preceding construct corresponds to

```
var
  ByteVar: Byte;
  WordVar: Word;
```

```
  ...
asm
  mov     al,ByteVar
  mov     bx,WordVar
end;
```

Operands

Inline assembler operands are expressions, which consist of a combination of constants, registers, symbols, and operators. Although inline assembler expressions are built using the same basic principles as Pascal expressions, there are a number of important differences, as will be explained in a following section.

Within operands, the following reserved words have a predefined meaning to the inline assembler:

AH	CL	FAR	SEG
AL	CS	HIGH	SHL
AND	CX	LOW	SHR
AX	DH	MOD	SI
BH	DI	NEAR	SP
BL	DL	NOT	SS
BP	DS	OFFSET	ST
BX	DWORD	OR	TBYTE
BYTE	DX	PTR	TYPE
CH	ES	QWORD	WORD
			XOR

The reserved words always take precedence over user-defined identifiers. For instance, the code fragment

```
var
  ch: Char;
  ...
asm
  mov     ch, 1
end;
```

will load 1 into the CH register, *not* into the *CH* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (**&**) identifier override operator:

```
asm
  mov     &ch, 1
end;
```

It is strongly suggested that you avoid user-defined identifiers with the same names as inline assembler reserved words, since such name confusion can easily lead to very obscure and hard-to-find bugs.

Expressions

The inline assembler evaluates all expressions as 32-bit integer values; it does not support floating-point and string values, except string constants.

Inline assembler expressions are built from *expression elements* and *operators*, and each expression has an associated *expression class* and *expression type*. These concepts are explained in the following sections.

Differences between Pascal and Assembler expressions

The most important difference between Pascal expressions and inline assembler expressions is that all inline assembler expressions must resolve to a *constant value*, in other words a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid inline assembler statement:

```
asm
  mov     Z,X+Y
end;
```

Since both *X* and *Y* are constants, the expression *X* + *Y* is merely a more convenient way of writing the constant 30, and the resulting instruction becomes a move immediate of the value 30 into the word-sized variable *Z*. But if you change *X* and *Y* to be variables,

```
var
  X, Y: Integer;
```

the inline assembler can no longer compute the value of X + Y at compile time. The correct inline assembler construct to move the sum of X and Y into Z now becomes

```
asm
  mov     ax,X
  add     ax,Y
  mov     Z,ax
end;
```

Another important difference between Pascal and inline assembler expressions is the way variables are interpreted. In a Pascal expression, a reference to a variable is interpreted as the *contents* of the variable, but in an inline assembler expression, a variable reference denotes the *address* of the variable. For example, in Pascal, the expression X + 4, where X is a variable, means the contents of X plus 4, whereas in the inline assembler it means the contents of the word at an address four bytes higher than the address of X. So, even though you're allowed to write

```
asm
  mov     ax,X+4
end;
```

the code does not load the value of X plus 4 into AX, but rather it loads the value of a word stored four bytes beyond X. The correct way to add 4 to the contents of X is:

```
asm
  MOV     AX,X
  ADD     AX,4
end;
```

Expression elements

The basic elements of an expression are *constants*, *registers*, and *symbols*.

Constants

The inline assembler supports two types of constants: *numeric constants* and *string constants*.

Numeric constants

Numeric constants must be integers, and their values must be between –2,147,483,648 and 4,294,967,295.

Numeric constants by default use decimal (base 10) notation, but the inline assembler supports binary (base 2), octal (base 8), and hexadecimal (base 16) notations as well. Binary notation is selected by writing a *B* after the number, octal notation is selected by writing a letter *O* after the number, and hexadecimal notation is selected by writing an *H* after the number or a $ before the number.

The *B*, *O*, and *H* suffixes are not supported in Pascal expressions. Pascal expressions allow only decimal notation (the default) and hexadecimal notation (using a $ prefix).

Numeric constants must start with one of the digits 0 through 9 or a $ character; thus, when you write a hexadecimal constant using the *H* suffix, an extra zero in front of the number is required if the first significant digit is one of the hexadecimal digits *A* through *F*. For example, 0BAD4H and $BAD4 are hexadecimal constants, but BAD4H is an identifier since it starts with a letter and not a digit.

String constants

String constants must be enclosed in single or double quotes. Two consecutive quotes of the same type as the enclosing quotes count as only one character. Here are some examples of string constants:

```
'Z'
'Turbo Pascal'
"That's all folks"
'"That"s all folks," he said.'
'100'
''''
""""
```

Notice in the fourth string the use of two consecutive single quotes to denote one single quote character.

String constants of any length are allowed in **DB** directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters, and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the

leftmost (first) character(s) are assumed to be zero. Some examples of string constants and their corresponding numeric values follow:

'a'	00000061H
'ba'	00006261H
'cba'	00636261H
'dcba'	64636261H
'a '	00006120H
' a'	20202061H
'a'*2	000000E2H
'a'-'A'	00000020H
not 'a'	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers:

16-bit general purpose	AX	BX	CX	DX
8-bit low registers	AL	BL	CL	DL
8-bit high registers	AH	BH	CH	DH
16-bit pointer or index	SP	BP	SI	DI
16-bit segment registers	CS	DS	SS	ES
8087 register stack	ST			

When an operand consists solely of a register name, it is called a register operand. All registers can be used as register operands. In addition, some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI].

The segment registers (ES, CS, SS, and DS) can be used in conjunction with the colon (:) segment override operator to indicate a different segment than the one the processor selects by default.

The symbol **ST** denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using **ST(*x*)**, where *x* is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The inline assembler allows you to access almost all Pascal symbols in assembler expressions, including labels, constants, types, variables, procedures, and functions. In addition, the inline assembler implements the following special symbols:

@Code *@Data* *@Result*

The *@Code* and *@Data* symbols represent the current code and data segments. They should only be used in conjunction with the **SEG** operator:

```
asm
  mov     ax,SEG @Data
  mov     ds,ax
end;
```

The *@Result* symbol represents the function result variable within the statement part of a function. For example, in the function

```
function Sum(X, Y: Integer): Integer;
begin
  Sum := X + Y;
end;
```

the statement that assigns a function result value to *Sum* would use the *@Result* variable if it was written in inline assembler:

```
function Sum(X, Y: Integer): Integer;
begin
  asm
    mov     ax,X
    add     ax,Y
    mov     @Result,AX
  end;
end;
```

The following symbols cannot be used in inline assembler expressions:

- Standard procedures and functions (for example, *WriteLn*, *Chr*).
- The *Mem*, *MemW*, *MemL*, *Port*, and *PortW* special arrays.
- String, floating-point, and set constants.
- Procedures and functions declared with the **inline** directive.
- Labels that aren't declared in the current block.
- The *@Result* symbol outside a function.

Table 22.1 summarizes the value, class, and type of the different kinds of symbols that can be used in inline assembler expressions. (Expression classes and types are described in a following section.)

Table 22.1
Values, classes, and types of symbols

Symbol	Value	Class	Type
Label	Address of label	Memory	SHORT
Constant	Value of constant	Immediate	0
Type	0	Memory	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory	Size of type
Procedure	Address of procedure	Memory	NEAR or FAR
Function	Address of function	Memory	NEAR or FAR
Unit	0	Immediate	0
@Code	Code segment address	Memory	0FFF0H
@Data	Data segment address	Memory	0FFF0H
@Result	Result var offset	Memory	Size of type

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to SS:BP, and the value of a local variable symbol is its signed offset from SS:BP. The assembler automatically adds [BP] in references to local variables. For example, given the declarations

```
procedure Test;
var
  Count: Integer;
```

the instruction

```
asm
  mov     ax,Count
end;
```

assembles into `MOV AX,[BP-2]`.

The inline assembler always treats a **var** parameter as a 32-bit pointer, and the size of a **var** parameter is always 4 (the size of a 32-bit pointer). In Pascal, the syntax for accessing a **var** parameter and a value parameter is the same—this is not the case in inline assembler. Since **var** parameters are really pointers, you have to treat them as such in inline assembler. So, to access the contents of a **var** parameter, you first have to load the 32-bit pointer and then access the location it points to. For example, if the *X* and *Y* parameters of the above function *Sum* were **var** parameters, the code would look like this:

```
function Sum(var X, Y: Integer): Integer;
begin
  asm
    les     bx,X
    mov     ax,es:[bx]
    les     bx,Y
    add     ax,es:[bx]
    mov     @Result,ax
  end;
end;
```

Some symbols, such as record types and variables, have a scope which can be accessed using the period (.) structure member selector operator. For example, given the declarations

```
type
  Point = record
    X, Y: Integer;
  end;
  Rect = record
    A, B: Point;
  end;
var
  P: Point;
  R: Rect;
```

the following constructs can be used to access fields in the *P* and *R* variables:

```
asm
  mov     ax,P.X
  mov     dx,P.Y
  mov     cx,R.A.X
  mov     bx,R.B.Y
end;
```

A type identifier can be used to construct variables "on the fly". Each of the instructions below generate the same machine code, which loads the contents of ES:[DI+4] into AX.

```
asm
  mov     ax,(Rect PTR es:[di]).B.X
  mov     ax,Rect(es:[di]).B.X
  mov     ax,es:Rect[di].B.X
  mov     ax,Rect[es:di].B.X
  mov     ax,es:[di].Rect.B.X
end;
```

A scope is provided by type, field, and variable symbols of a record or object type. In addition, a unit identifier opens the scope of a particular unit, just like a fully qualified identifier in Pascal.

Expression classes

The inline assembler divides expressions into three classes: *registers, memory references,* and *immediate values.*

An expression that consists solely of a register name is a register expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references; Pascal's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values; this group includes Pascal's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;

...

asm
  mov     ax,Start            { MOV AX,xxxx }
  mov     bx,Count            { MOV BX,[xxxx] }
  mov     cx,[Start]          { MOV CX,[xxxx] }
  mov     dx,OFFSET Count     { MOV DX,xxxx }
end;
```

Since *Start* is an immediate value, the first **MOV** is assembled into a move immediate instruction. The second **MOV**, however, is translated into a move memory instruction, as *Count* is a memory reference. In the third **MOV**, the square brackets operator is used to convert *Start* into a memory reference (in this case, the word at offset 10 in the data segment), and in the fourth **MOV**, the **OFFSET** operator is used to convert *Count* into an immediate value (the offset of *Count* in the data segment).

As you can see, the square brackets and the **OFFSET** operators complement each other. In terms of the resulting machine code, the following **asm** statement is identical to the first two lines of the previous **asm** statement:

```
asm
  mov     ax,OFFSET [Start]
  mov     bx,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either *relocatable expressions* or *absolute expressions*. A relocatable expression denotes a value that requires *relocation* at link time, and an absolute expression denotes a value that requires no such relocation. Typically, an expression that refers to a label, variable, procedure, or function is relocatable, and an expression that operates solely on constants is absolute.

Relocation is the process by which the linker assigns absolute addresses to symbols. At compile time, the compiler does not know the final address of a label, variable, procedure, or function; it does not become known until link time, when the linker assigns a specific absolute address to the symbol.

The inline assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression types

Every inline assembler expression has an associated type—or more correctly, an associated size, since the inline assembler regards the type of an expression simply as the size of its memory location. For example, the type (size) of an *Integer* variable is two, since it occupies 2 bytes.

The inline assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;

...

asm
  mov     al,QuitFlag
  mov     bx,OutBufPtr
end;
```

the inline assembler checks that the size of *QuitFlag* is one (a byte), and that the size of *OutBufPtr* is two (a word). An error results if the type check fails; for example, the following is not allowed:

```
asm
  mov     dl,OutBufPtr
end;
```

since DL is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; correct ways of writing the previous instruction are

```
asm
  mov     dl,BYTE PTR OutBufPtr
  mov     dl,Byte(OutBufPtr)
  mov     dl,OutBufPtr.Byte
end;
```

all of which refer to the first (least significant) byte of the *OutBufPtr* variable.

In some cases, a memory reference is untyped, that is, it has no associated type. One example is an immediate value enclosed in square brackets:

```
asm
  mov     al,[100H]
  mov     bx,[100H]
end;
```

The inline assembler permits both of these instructions, since the expression [100H] has no associated type—it just means "the contents of address 100H in the data segment," and the type can be determined from the first operand (byte for AL, word for BX). In cases where the type cannot be determined from another operand, the inline assembler requires an explicit typecast:

```
asm
  inc     BYTE PTR [100H]
  imul    WORD PTR [100H]
end;
```

Table 22.2 summarizes the predefined type symbols that the inline assembler provides in addition to any currently declared Pascal types.

Table 22.2
Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	0FFFEH
FAR	0FFFFH

Notice in particular the **NEAR** and **FAR** pseudo-types, which are used by procedure and function symbols to indicate their call model. You can use **NEAR** and **FAR** in typecasts just like other symbols. For example, if *FarProc* is a **FAR** procedure,

```
procedure FarProc; far;
```

and if you are writing inline assembler code in the same module as *FarProc*, you can use the more efficient **NEAR** call instruction to call it:

```
asm
  push    cs
  call    NEAR PTR FarProc
end;
```

Expression operators

The inline assembler provides a variety of operators, divided into 12 classes of precedence. Table 22.3 lists the inline assembler's expression operators in decreasing order of precedence.

Table 22.3
Inline asssembler expression operators

Operator(s)	Comments
&	Identifier override operator
(), [], .	Structure member selector
HIGH, LOW	
+, –	Unary operators
:	Segment override operator
OFFSET, SEG, TYPE, PTR, *, /, MOD, SHL, SHR	
+, –	Binary addition/ subtraction operators
NOT, AND, OR, XOR	Bitwise operators

*Inline assembler operator precedence is different from Pascal. For example, in an inline assembler expression, the **AND** operator has lower precedence than the plus (+) and minus (-) operators, whereas in a Pascal expression, it has higher precedence.*

&	**Identifier override.** The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as an inline assembler reserved symbol.
(...)	**Subexpression.** Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression may optionally precede the expression within the parentheses; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference.** The expression within brackets is evaluated completely prior to being treated as a single expression element. The expression within brackets may be combined with the BX, BP, SI, or DI registers using the plus (+) operator, to indicate CPU register indexing. Another expression may optionally precede the expression within the brackets; the result in this case becomes the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector.** The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
+	**Unary plus.** Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
–	**Unary minus.** Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.

:	**Segment override.** Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction will be prefixed by an appropriate segment override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (low-order word) of the expression following the operator. The result is an immediate value.
SEG	Returns the segment part (high-order word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator.** The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	**Multiplication.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
+	**Addition.** The expressions can be immediate values or memory references, but only one of the expressions

can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions are memory references, the result is also a memory reference.

–	**Subtraction.** The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR.** Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembler procedures and functions

So far, every **asm**...**end** construct you've seen has been a statement within a normal **begin**...**end** statement part. Turbo Pascal's assembler directive allows you to write complete procedures and functions in inline assembler, without the need for a **begin**...**end** statement part. Here's an example of an assembler function:

```
function LongMul(X, Y: Integer): Longint; assembler;
asm
  mov     ax,X
  imul    Y
end;
```

The assembler directive causes Turbo Pascal to perform a number of code generation optimizations:

- The compiler doesn't generate code to copy value parameters into local variables. This affects all string-type value param-

eters, and other value parameters whose size is not 1, 2, or 4 bytes. Within the procedure or function, such parameters must be treated as if they were **var** parameters.

- The compiler doesn't allocate a function result variable, and a reference to the *@Result* symbol is an error. String functions, however, are an exception to this rule—they always have a *@Result* pointer which gets allocated by the caller.
- The compiler generates no stack frame for procedures and functions that have no parameters and no local variables.
- The automatically generated entry and exit code for an assembler procedure or function looks like this:

```
push    bp              ;Present if Locals <> 0 or Params <> 0
mov     bp,sp           ;Present if Locals <> 0 or Params <> 0
sub     sp,Locals       ;Present if Locals <> 0
...
mov     sp,bp           ;Present if Locals <> 0
pop     bp              ;Present if Locals <> 0 or Params <> 0
ret     Params          ;Always present
```

  where *Locals* is the size of the local variables, and *Params* is the size of the parameters. If both *Locals* and *Params* are zero, there is no entry code, and the exit code consists simply of a **RET** instruction.

Functions using the assembler directive must return their results as follows:

- Ordinal-type function results (Integer, Char, Boolean, and enumerated types) are returned in AL (8-bit values), AX (16-bit values), or DX:AX (32-bit values).
- Real-type function results (type Real) are returned in DX:BX:AX.
- 8087-type function results (type Single, Double, Extended, and Comp) are returned in ST(0) on the 8087 coprocessor's register stack.
- Pointer-type function results are returned in DX:AX.
- String-type function results are returned in the temporary location pointed to by the *@Result* function result symbol.

The assembler directive is in many ways comparable to the **external** directive, and assembler procedures and functions must obey the same rules as **external** procedures and functions. The following examples demonstrate some of the differences between **asm** statements in normal functions and assembler functions. The

first example uses an **asm** statement in a normal function to convert a string to upper case. Notice that the value parameter *Str* in this case refers to a local variable, since the compiler automatically generates entry code that copies the actual parameter into local storage.

```
function UpperCase(Str: String): String;
begin
  asm
    cld
    lea     si,Str
    les     di,@Result
    SEGSS   lodsb
    stosb
    xor     ah,ah
    xchg    ax,cx
    jcxz    @3
  @1:
    SEGSS   lodsb
    cmp     al,'a'
    jb      @2
    cmp     al,'z'
    ja      @2
    sub     al,20H
  @2:
    stosb
    loop    @1
  @3:
  end;
end;
```

The second example is an assembler version of the *UpperCase* function. In this case, *Str* is not copied into local storage, and the function must treat *Str* as a **var** parameter.

```
function UpperCase(S: String): String; assembler;
asm
  push    ds
  cld
  lds     si,Str
  les     di,@Result
  lodsb
  stosb
  xor     ah,ah
  xchg    ax,cx
  jcxz    @3
@1:
  lodsb
```

```
    cmp     al,'a'
    jb      @2
    cmp     al,'z'
    ja      @2
    sub     al,20H
@2:
    stosb
    loop    @1
@3:
    pop     ds
end;
```

C H A P T E R

23

Linking assembler code

Procedures and functions written in assembly language can be linked with Turbo Pascal programs or units using the **$L** compiler directive. The assembly language source file must be assembled into an object file (extension .OBJ) using an assembler like Turbo Assembler. Multiple object files can be linked with a program or unit through multiple **$L** directives.

Procedures and functions written in assembly language must be declared as **external** in the Pascal program or unit, for example,

```
function LoCase(Ch: Char): Char; external;
```

In the corresponding assembly language source file, all procedures and functions must be placed in a segment named "**CODE**" or "**CSEG**", or in a segment whose name ends in **_TEXT**, and the names of the external procedures and functions must appear in **PUBLIC** directives.

You must ensure that an assembly language procedure or function matches its Pascal definition with respect to call model (near or far), number of parameters, types of parameters, and result type.

An assembly language source file can declare initialized variables in a segment named **CONST** or in a segment whose name ends in **_DATA**, and uninitialized variables in a segment named **DATA** or **DSEG**, or in a segment whose name ends in **_BSS**. Such variables are private to the assembly language source file and cannot be referenced from the Pascal program or unit. However, they reside

in the same segment as the Pascal globals, and can be accessed through the DS segment register.

All procedures, functions, and variables declared in the Pascal program or unit, and the ones declared in the **interface** section of the used units, can be referenced from the assembly language source file through **EXTRN** directives. Again, it is up to you to supply the correct type in the **EXTRN** definition.

When an object file appears in a **$L** directive, Turbo Pascal converts the file from the Intel relocatable object module format (.OBJ) to its own internal relocatable format. This conversion is possible only if certain rules are observed:

- All procedures and functions must be placed in a segment named **CODE** or **CSEG**, or in a segment with a name that ends in **_TEXT**. All initialized private variables must be placed in a segment named **CONST**, or in a segment with a name that ends in **_DATA**. All uninitialized private variables must be placed in a segment named **DATA** or **DSEG**, or in a segment with a name that ends in **_BSS**. All other segments are ignored, and so are **GROUP** directives. The segment definitions can specify **BYTE** or **WORD** alignment, but when linked, code segments are always byte aligned, and data segments are always word aligned. The segment definitions can optionally specify **PUBLIC** and a class name, both of which are ignored.
- Turbo Pascal ignores any data for segments other than the code segment (**CODE**, **CSEG**, or *xxxx*_**TEXT**) and the initialized data segment (**CONST** or *xxxx*_**DATA**). So, when declaring variables in the uninitialized data segment (**DATA**, **DSEG**, or *xxxx*_**BSS**), always use a question mark (?) to specify the value, for instance:

```
Count   DW  ?
Buffer  DB  128 DUP(?)
```

- Byte-sized references to **EXTRN** symbols are not allowed. For example, this means that the assembly language **HIGH** and **LOW** operators cannot be used with **EXTRN** symbols.

Turbo Assembler and Turbo Pascal

Turbo Assembler (TASM) makes it easy to program routines in assembly language and interface them into your Turbo Pascal programs. Turbo Assembler provides simplified segmentation and language support for Pascal programmers.

The **.MODEL** directive specifies the memory model for an assembler module that uses simplified segmentation. For linking with Pascal programs, the **.MODEL** syntax looks like this:

```
.MODEL xxxx, PASCAL
```

xxxx is the memory model (usually this is large).

Specifying the language PASCAL in the **.MODEL** directive tells Turbo Assembler that the arguments were pushed onto the stack from left to right, in the order they were encountered in the source statement that called the procedure.

The **PROC** directive lets you define your parameters in the same order as they are defined in your Pascal program. If you are defining a function that returns a string, notice that the **PROC** directive has a **RETURNS** option that lets you access the temporary string pointer on the stack without affecting the number of parameter bytes added to the **RET** statement.

Here's an example coded to use the **.MODEL** and **PROC** directives:

```
        .MODEL large, PASCAL
        .CODE
MyProc PROC   FAR I : BYTE, J : BYTE RETURNS Result : DWORD
        PUBLIC MyProc
        les    DI, Result       ;get address of temporary string
        mov    AL, I            ;get first parameter I
        mov    BL, J            ;get second parameter J
        .
        .
        .
        ret
```

The Pascal function definition would look like this:

```
function MyProc(I, J : Char) : string; external;
```

For more information about interfacing Turbo Assembler with Turbo Pascal, refer to Chapter 6 of the *Turbo Assembler User's Guide*.

Examples of assembly language routines

The following code is an example of a unit that implements two assembly language string-handling routines. The *UpperCase* function converts all characters in a string to uppercase, and the *StringOf* function returns a string of characters of a specified length.

```
unit Stringer;
interface
function UpperCase(S: String): String;
function StringOf(Ch: Char; Count: Byte): String;
implementation
{$L STRS}
function UpperCase; external;
function StringOf; external;
end.
```

The assembly language file that implements the *UpperCase* and *StringOf* routines is shown next. It must be assembled into a file called STRS.OBJ before the *Stringer* unit can be compiled. Note that the routines use the far call model because they are declared in the **interface** section of the unit. This example uses standard segmentation.

```
CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE
        PUBLIC  UpperCase, StringOf     ;Make them known

; function UpperCase(S: String): String

UpperRes        EQU     DWORD PTR [BP + 10]
UpperStr        EQU     DWORD PTR [BP + 6]

UpperCase       PROC FAR

        push    bp                      ;Save BP
        mov     bp, sp                  ;Set up stack frame
        push    ds                      ;Save DS
        lds     si, Upperstr            ;Load string address
        les     di, Upperres            ;Load result address
        cld                             ;Forward string-ops
        lodsb                           ;Load string length
        stosb                           ;Copy to result
        mov     cl, al                  ;String length to CX
        xor     ch, ch
        jcxz    U3                      ;Skip if empty string
U1:     lodsb                           ;Load character
        cmp     al, 'a'                 ;Skip if not 'a'..'z'
        jb      U2
        cmp     al, 'z'
        ja      U2
        sub     al, 'a'-'a'             ;Convert to uppercase
U2:     stosb                           ;Store in result
        loop    U1                      ;Loop for all characters
U3:     pop     ds                      ;Restore DS
        pop     bp                      ;Restore BP
        ret     4                       ;Remove parameter and return
```

```
UpperCase       ENDP

; procedure StringOf(var S: String; Ch: Char; Count: Byte)

StrOfS          EQU     DWORD PTR [BP + 10]
StrOfChar       EQU     BYTE PTR [BP + 8]
StrOfCount      EQU     BYTE PTR [BP + 6]
StringOf        PROC FAR
        push    bp                      ;Save BP
        mov     bp, sp                  ;Set up stack frame
        les     di, StrOfRes            ;Load result address
        mov     al, StrOfCount          ;Load count
        cld                             ;Forward string-ops
        stosb                           ;Store length
        mov     cl, al                  ;Count to CX
        xor     ch, ch
        mov     al, StrOfChar           ;Load character
        rep     STOSB                   ;Store string of characters
        pop     bp                      ;Restore BP
        ret     8                       ;Remove parameters and return

StringOf        ENDP
CODE    ENDS
        END
```

To assemble the example and compile the unit, use the following commands:

```
TASM STR5
TPCW stringer
```

The next example shows how an assembly language routine can refer to Pascal routines and variables. The *Numbers* program reads up to 100 Integer values, and then calls an assembly language procedure to check the range of each of these values. If a value is out of range, the assembly language procedure calls a Pascal procedure to print it.

```
program Numbers;
uses WinCrt;
{$L CHECK}
var
  Buffer: array[1..100] of Integer;
  Count: Integer;

procedure RangeError(N: Integer);
begin
  Writeln('Range error: ',N);
end;

procedure CheckRange(Min, Max: Integer); external;
begin
```

```
  Count := 0;
  while not Eof and (Count < 100) do
  begin
  { Ends when you type Ctrl-Z or after 100 iterations }
    Count := Count + 1;
    Readln(Buffer[Count]);
  end;
  CheckRange(-10, 10);
end.
```

The assembly language file that implements the *CheckRange* procedure is shown next. It must be assembled into a file called CHECK.OBJ before the *Numbers* program can be compiled. Note that the procedure uses the near call model because it is declared in a program. This example uses standard segmentation.

```
DATA    SEGMENT WORD PUBLIC
        EXTRN   Buffer : WORD, Count : WORD ;Pascal variables
DATA    ENDS
CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS : CODE, DS : Buffer
        EXTRN   RangeError : NEAR       ;Implemented in Pascal
        PUBLIC  CheckRange              ;Implemented here

CheckRange      PROC    NEAR

        mov     bx,sp                   ;Get parameters pointer
        mov     ax, ss:[BX + 4]         ;Load Min
        mov     dx, ss:[BX + 2]         ;Load Max
        xor     bx, bx                  ;Clear Data index
        mov     cx, count               ;Load Count
        jcxz    SD4                     ;Skip if zero
SD1:    cmp     Buffer[BX], AX          ;Too small?
        jl      SD2                     ;Yes, jump
        cmp     Buffer[BX], DX          ;Too large?
        jle     SD3                     ;No, jump
SD2:    push    ax                      ;Save registers
        push    bx
        push    cx
        push    dx
        push    Buffer[BX]              ;Pass offending value to
                                        ; Pascal
        call    RangeError              ;Call Pascal procedure
        pop     dx                      ;Restore registers
        pop     cx
        pop     bx
        pop     ax
SD3:    inc     bx                      ;Point to next element
        inc     bx
        loop    SD1                     ;Loop for each item
```

```
SD4:    ret     4                       ;Clean stack and return

CheckRange      ENDP
CODE    ENDS
        END
```

Turbo Assembler example

Here's a Turbo Assembler version of the previous assembly language example uses simplified segmentation directives:

```
        .MODEL  large, PASCAL
        LOCALS  @@                      ;Define local labels
                                        ; prefix
        .DATA                           ;Data segment
        EXTRN   Buffer : WORD, Count : WORD
                                        ;Pascal variables

        .CODE                           ;Code segment
        EXTRN   RangeError : NEAR       ;Implemented in Pascal
        PUBLIC  CheckRange              ;Implemented here

CheckRange      PROC NEAR Min : WORD, Max : WORD

        mov     ax, Min                 ;Keep Min in AX
        mov     dx, Max                 ;Keep Max in DX
        xor     bx, BX                  ;Clear Buffer index
        mov     cx, Count               ;Load Count
        jcxz    @@4                     ;Skip if zero
@@1:    cmp     ax, Buffer[BX]          ;Too small?
        jg      @@2                     ;Yes, go to CR2
        cmp     dx, Buffer[BX]          ;Too large?
        jge     @@3                     ;No, go to CR3
@@2:    push    ax                      ;Save registers
        push    bx
        push    cx
        push    dx
        push    Buffer[BX]              ;Pass offending value to
                                        ; Pascal
        call    RangeError              ;Call Pascal procedure
        pop     dx                      ;Restore registers
        pop     cx
        pop     bx
        pop     ax
@@3:    inc     bx                      ;Point to next element
        inc     bx
        loop    @@1                     ;Loop for each item
@@4:    ret                             ;Done

CheckRange      ENDP
        END
```

Inline machine code

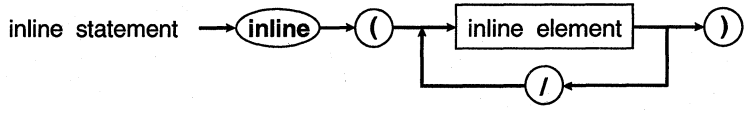
For very short assembly language subroutines, Turbo Pascal's **inline** statements and directives are very convenient. They let you insert machine code instructions directly into the program or unit text instead of through an object file.

Inline statements

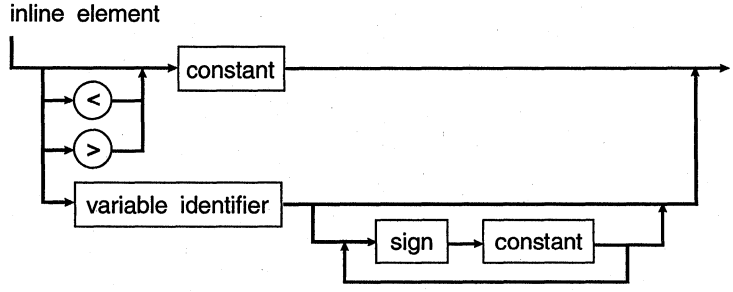
An **inline** statement consists of the reserved word **inline** followed by one or more inline elements, separated by slashes and enclosed in parentheses:

```
inline(10/$2345/Count + 1/Data - Offset);
```

Here's the syntax of an inline statement:



Each inline element consists of an optional size specifier, **<** or **>**, and a constant or a variable identifier, followed by zero or more offset specifiers (see the syntax that follows). An offset specifier consists of a **+** or a **–** followed by a constant.



Each inline element generates 1 byte or one word of code. The value is computed from the value of the first constant or the offset of the variable identifier, to which is added or subtracted the value of each of the constants that follow it.

An inline element generates 1 byte of code if it consists of constants only and if its value is within the 8-bit range (0..255). If the value is outside the 8-bit range or if the inline element refers to a variable, one word of code is generated (least-significant byte first).

The < and > operators can be used to override the automatic size selection we described earlier. If an inline element starts with a < operator, only the least-significant byte of the value is coded, even if it is a 16-bit value. If an inline element starts with a > operator, a word is always coded, even though the most-significant byte is 0. For example, the statement

```
inline(<$1234/>$44);
```

generates 3 bytes of code: $34, $44, $00.

*Registers BP, SP, SS, and DS must be preserved by **inline** statements; all other registers can be modified.*

The value of a variable identifier in an inline element is the offset address of the variable within its base segment. The base segment of global variables—variables declared at the outermost level in a program or a unit—and typed constants is the data segment, which is accessible through the DS register. The base segment of local variables—variables declared within the current subprogram—is the stack segment. In this case the variable offset is relative to the BP register, which automatically causes the stack segment to be selected.

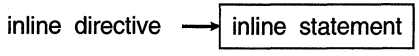
The following example of an **inline** statement generates machine code for storing a specified number of words of data in a specified variable. When called, procedure *FillWord* stores *Count* words of the value *Data* in memory, starting at the first byte occupied by *Dest*.

```
procedure FillWord(var Dest; Count, Data: Word);
begin
  inline(
    $C4/$BE/Dest/                    { LES DI,Dest[BP] }
    $8B/$8E/Count/                   { MOV CX,Count[BP] }
    $8B/$86/Data/                    { MOV AX,Data[BP] }
    $FC/                             { CLD }
    $F3/$AB);                        { REP STOSW }
end;
```

Inline statements can be freely mixed with other statements throughout the statement part of a block.

Inline directives

Inline directives let you write procedures and functions that expand into a given sequence of machine code instructions whenever they are called. These are comparable to macros in assembly language. The syntax for an inline directive is the same as that of an inline statement:



When a normal procedure or function is called (including one that contains **inline** statements), the compiler generates code that pushes the parameters (if any) onto the stack, and then generates a **CALL** instruction to call the procedure or function. However, when you call an inline procedure or function, the compiler generates code from the inline directive instead of the **CALL**. Here's a short example of two inline procedures:

```
procedure DisableInterrupts; inline($FA);       { CLI }
procedure EnableInterrupts; inline($FB);        { STI }
```

When *DisableInterrupts* is called, it generates 1 byte of code—a **CLI** instruction.

Procedures and functions declared with inline directives can have parameters; however, the parameters cannot be referred to symbolically in the inline directive (other variables can, though). Also, because such procedures and functions are in fact macros, there is no automatic entry and exit code, nor should there be any return instruction.

The following function multiplies two Integer values, producing a Longint result:

```
function LongMul(X, Y: Integer): Longint;
inline(
  $5A/                              { POP AX ;Pop X }
  $58/                              { POP DX ;Pop Y }
  $F7/$EA);                         { IMUL DX ;DX : AX = X * Y }
```

Note the lack of entry and exit code and the missing return instruction. These are not required, because the 4 bytes are inserted into the instruction stream when *LongMul* is called.

Inline directives are intended for very short (less than 10 bytes) procedures and functions only.

Because of the macro-like nature of inline procedures and functions, they cannot be used as arguments to the **@** operator and the *Addr*, *Ofs*, and *Seg* functions.

PART 5

Library reference

C H A P T E R

24

The run-time library

This chapter contains a detailed description of all the procedures and functions in Turbo Pascal. The following sample library lookup entry explains where to look for details about each Turbo Pascal procedure and function.

Sample procedure — Unit it occupies

Function What it does

Declaration How it's declared; italicized items are user-defined

Result type What it returns if it's a function

Remarks General information about the procedure or function

Restrictions Special requirements or items to watch for

See also Related procedures and functions

Example Here you'll find a sample program that shows the use of the procedure or function in that entry. All of the examples that produce output (Writeln) or require input (Readln) should use the *WinCrt* unit.

Abs function

Function Returns the absolute value of the argument.

Declaration `Abs(X)`

Result type Same type as parameter.

Remarks *X* is an integer-type or real-type expression. The result, of the same type as *X*, is the absolute value of *X*.

Example

```
var
  r: Real;
  i: Integer;
begin
  r := Abs(-2.3);        { 2.3 }
  i := Abs(-157);        { 157 }
end.
```

Addr function

Function Returns the address of a specified object.

Declaration `Addr(X)`

Result type Pointer

Remarks *X* is any variable, or a procedure or function identifier. The result is a pointer that points to *X*. Like **nil**, the result of *Addr* is assignment compatible with all pointer types.

The @ operator produces the same result as *Addr*.

See also *Ofs, Ptr, Seg*

Example

```
var
  P: Pointer;
begin
  P := Addr(P);          { Now points to itself }
end.
```

Append procedure

Function Opens an existing file for appending.

Declaration `Append(var F: Text)`

Remarks *F* is a text-file variable that must have been associated with an external file using *Assign*.

Append opens the existing external file with the name assigned to *F*. It is an error if there is no existing external file of the given name. If *F* was already open, it is first closed and then re-opened. The current file position is set to the end of the file.

If a *Ctrl+Z* (ASCII 26) is present in the last 128-byte block of the file, the current file position is set to overwrite the first *Ctrl+Z* in the block. In this way, text can be appended to a file that terminates with a *Ctrl+Z*.

If *F* was assigned an empty name, such as *Assign*(*F*, ''), then, after the call to *Append*, *F* will refer to the standard output file (standard handle number 1).

After a call to *Append*, *F* becomes write-only, and the file pointer is at end-of-file.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Assign, Close, Reset, Rewrite*

Example

```
var F: Text;
begin
  Assign(F, 'TEST.TXT');
  Rewrite(F);                                    { Create new file }
  Writeln(F, 'original text');
  Close(F);                                  { Close file, save changes }
  Append(F);                                   { Add more text onto end }
  Writeln(F, 'appended text');
  Close(F);                                  { Close file, save changes }
end.
```

ArcTan function

Function Returns the arctangent of the argument.

Declaration `ArcTan(x: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the principal value, in radians, of the arctangent of *X*.

See also *Cos*, *Sin*

Example

```
var
  R: Real;
begin
  R := ArcTan(Pi);
end.
```

Assign procedure

Function Assigns the name of an external file to a file variable.

Declaration `Assign(var F; Name)`

Remarks *F* is a file variable of any file type, and *Name* is a string-type expression or an expression of type *PChar* if the extended syntax is enabled. All further operations on *F* will operate on the external file with the file name *Name*.

After a call to *Assign*, the association between *F* and the external file continues to exist until another *Assign* is done on *F*.

A file name consists of a path of zero or more directory names separated by backslashes, followed by the actual file name:

```
Drive:\DirName\...\DirName\FileName
```

If the path begins with a backslash, it starts in the root directory; otherwise, it starts in the current directory.

Drive is a disk drive identifier (*A*-*Z*). If *Drive* and the colon are omitted, the default drive is used. *\DirName\...\DirName* is the root directory and subdirectory path to the file name. *FileName* consists of a name of up to eight characters, optionally followed by a period and an extension of up to three characters.

The maximum length of the entire file name is 79 characters.

A special case arises when *Name* is an empty string; that is, when *Length*(*Name*) is zero. In that case, *F* becomes associated with the standard input or standard output file. These special files allow a program to utilize the I/O redirection feature of the DOS operating system. If assigned an empty name, then after a call to *Reset*(*F*), *F* will refer to the standard input file, and after a call to *Rewrite*(*F*), *F* will refer to the standard output file.

Restrictions *Assign* must never be used on an open file.

See also *Append, Close, Reset, Rewrite*

Example

```
{ Try redirecting this program from DOS to PRN, disk file, etc. }
var F: Text;
begin
  Assign(F, '');                                    { Standard output }
  Rewrite(F);
  Writeln(F, 'standard output...');
  Close(F);
end.
```

AssignCrt procedure — WinCrt

Function Assocates a text file with the CRT window.

Declaration

```
AssignCrt(var F: Text)
```

Remarks *AssignCrt* works exactly like the *Assign* standard procedure, except that no file name is specified. Instead, the text file is associated with the CRT window. Subsequent *Write* and *WriteLn* operations on the file will write to the CRT window, and *Read* and *ReadLn* operations will read from the CRT window.

BlockRead procedure

Function Reads one or more records into a variable.

Declaration

```
BlockRead(var F: file; var Buf; Count: Word [ ; var Result: Word ] )
```

Remarks *F* is an untyped file variable, *Buf* is any variable, *Count* is an expression of type Word, and *Result* is a variable of type Word.

BlockRead reads *Count* or less records from the file *F* into memory, starting at the first byte occupied by *Buf*. The actual number of complete records read (less than or equal to *Count*) is returned in the optional parameter

Result. If *Result* is not specified, an I/O error will occur if the number read is not equal to *Count*.

The entire block transferred occupies at most *Count* * *RecSize* bytes, where *RecSize* is the record size specified when the file was opened (or 128 if it was omitted). It's an error if *Count* * *RecSize* is greater than 65,535 (64K).

Result is an optional parameter. Here is how it works: If the entire block was transferred, *Result* will be equal to *Count* on return. Otherwise, if *Result* is less than *Count*, the end of the file was reached before the transfer was completed. In that case, if the file's record size is greater than one, *Result* returns the number of complete records read; that is, a possible last partial record is not included in *Result*.

The current file position is advanced by *Result* records as an effect of the *BlockRead*.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions File must be open.

See also *BlockWrite*

Example

```
program CopyFile;
{ Simple, fast file copy program with NO error-checking }
var
  FromF, ToF: file;
  NumRead, NumWritten: Word;
  Buf: array[1..2048] of Char;
begin
  Assign(FromF, ParamStr(1));                          { Open input file }
  Reset(FromF, 1);                                     { Record size = 1 }
  Assign(ToF, ParamStr(2));                           { Open output file }
  Rewrite(ToF, 1);                                     { Record size = 1 }
  Writeln('Copying ', FileSize(FromF), ' bytes...');
  repeat
    BlockRead(FromF, Buf, SizeOf(Buf), NumRead);
    BlockWrite(ToF, Buf, NumRead, NumWritten);
  until (NumRead = 0) or (NumWritten <> NumRead);
  Close(FromF);
  Close(ToF);
end.
```

BlockWrite procedure

Function Writes one or more records from a variable.

Declaration

```
BlockWrite(BlockWrite(var F:file; var Buf; Count: Word [ ; var Result: Word ] )
```

Remarks *F* is an untyped file variable, *Buf* is any variable, *Count* is an expression of type Word, and *Result* is a variable of type Word.

BlockWrite writes *Count* or less records to the file *F* from memory, starting at the first byte occupied by *Buf*. The actual number of complete records written (less than or equal to *Count*) is returned in the optional parameter *Result*. If *Result* is not specified, an I/O error will occur if the number written is not equal to *Count*.

The entire block transferred occupies at most *Count* * *RecSize* bytes, where *RecSize* is the record size specified when the file was opened (or 128 if it was omitted). It is an error if *Count* * *RecSize* is greater than 65,535 (64K).

Result is an optional parameter. Here is how it works: If the entire block was transferred, *Result* will be equal to *Count* on return. Otherwise, if *Result* is less than *Count*, the disk became full before the transfer was completed. In that case, if the file's record size is greater than one, *Result* returns the number of complete records written; that is, it's possible a remaining partial record is not included in *Result*.

The current file position is advanced by *Result* records as an effect of the *BlockWrite*.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions File must be open.

See also *BlockRead*

Example See example for *BlockRead*.

ChDir procedure

Function Changes the current directory.

Declaration
```
ChDir(S: String)
```

Remarks *S* is a string-type expression. The current directory is changed to a path specified by *S*. If *S* specifies a drive letter, the current drive is also changed.

With **{$I-}**, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *GetDir, MkDir, RmDir. SetCurDir* peforms the same function, but it takes a null-terminated string as an argument rather than a Pascal-style string.

Example
```
begin
  {$I-}
  { Get directory name from command line }
  ChDir(ParamStr(1));
  if IOResult <> 0 then
    Writeln('Cannot find directory');
end.
```

Chr function

Function Returns a character with a specified ordinal number.

Declaration
```
Chr(X: Byte)
```

Result type Char

Remarks *X* is an integer-type expression. The result is the character with an ordinal value (ASCII value) of X.

See also *Ord*

Example
```
uses WinCrt;
var
  I: Integer;
begin
  for I := 32 to 255 do Write(Chr(I));
end.
```

Close procedure

Function Closes an open file.

Declaration `Close(var F)`

Remarks *F* is a file variable of any file type that was previously opened with *Reset*, *Rewrite*, or *Append*. The external file associated with *F* is completely updated and then closed, and its DOS file handle is freed for reuse.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Append*, *Assign*, *Reset*, *Rewrite*

Example

```
var F: file;
begin
  Assign(F, '\AUTOEXEC.BAT');                    { Open file }
  Reset(F, 1);
  Writeln('File size = ', FileSize(F));
  Close(F);                                      { Close file }
end.
```

ClrEol procedure — WinCrt

Function Clears all characters from the cursor position to the end of the line without moving the cursor.

Declaration `ClrEol`

See also *ClrScr*

ClrScr procedure — WinCrt

Function Clears the screen.

Declaration `ClrScr`

Remarks When the screen is cleared the cursor returns to the upper left corner.

See also *ClrEol*

Concat function

Function Concatenates a sequence of strings.

Declaration `Concat(S1 [ , S2, ..., SN ]: String)`

Result type String

Remarks Each parameter is a string-type expression. The result is the concatenation of all the string parameters. If the resulting string is longer than 255 characters, it is truncated after the 255th character. Using the plus (+) operator returns the same results as using the *Concat* function:

```
S := 'ABC' + 'DEF';
```

See also *Copy, Delete, Insert, Length, Pos*

Example

```
var
  S: String;
begin
  S := Concat('ABC', 'DEF');                    { 'ABCDEF' }
end.
```

Copy function

Function Returns a substring of a string.

Declaration `Copy(S: String; Index: Integer; Count: Integer)`

Result type String

Remarks *S* is a string-type expression. *Index* and *Count* are integer-type expressions. *Copy* returns a string containing *Count* characters starting with the *Index*th character in *S*. If *Index* is larger than the length of *S*, an empty string is returned. If *Count* specifies more characters than remain starting at the *Index*th position, only the remainder of the string is returned.

See also *Concat, Delete, Insert, Length, Pos*

Example

```
var S: String;
begin
  S := 'ABCDEF';
  S := Copy(S, 2, 3)                            { 'BCD' }
end.
```

Cos function

Function Returns the cosine of the argument.

Declaration `Cos(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the cosine of *X*. *X* is assumed to represent an angle in radians.

See also *ArcTan, Sin*

Example

```
var R: Real;
begin
  R := Cos(Pi);
end.
```

CreateDir procedure WinDos

Function Creates a new subdirectory.

Declaration `CreateDir(Dir: PChar)`

Remarks The subdirectory to be created is specified in *Dir*. Errors are reported in *DosError*. *MkDir* performs the same function as *CreateDir*, but it takes a Pascal-style string as an argument rather than a null-terminated string.

See also *GetCurDir, SetCurDir, RemoveDir*

CSeg function

Function Returns the current value of the CS register.

Declaration `CSeg`

Result type Word

Remarks The result of type Word is the segment address of the code segment within which *CSeg* was called.

See also *DSeg, SSeg*

CursorTo procedure — WinCrt

Function Moves the cursor to the given coordinates within the virtual screen.

Declaration
```
CursorTo(X, Y: Integer)
```

Remarks The upper left corner corresponds to (0, 0). The *Cursor* variable is set to (*X*, *Y*).

Dec procedure

Function Decrements a variable.

Declaration
```
Dec(var X [ ; N: Longint ] )
```

Remarks *X* is an ordinal-type variable or a variable of type *PChar* if the extended syntax is enabled, and *N* is an integer-type expression. *X* is decremented by 1, or by *N* if *N* is specified; that is, *Dec*(*X*) corresponds to $X := X - 1$, and *Dec*(*X*, *N*) corresponds to $X := X - N$.

Dec generates optimized code and is especially useful in a tight loop.

See also *Inc, Pred, Succ*

Example
```
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Dec(IntVar);                              { IntVar := IntVar - 1 }
  Dec(LongintVar, 5);                   { LongintVar := LongintVar - 5 }
end.
```

Delete procedure

Function Deletes a substring from a string.

Declaration
```
Delete(var S: String; Index: Integer; Count: Integer)
```

Remarks *S* is a string-type variable. *Index* and *Count* are integer-type expressions. *Delete* deletes *Count* characters from *S* starting at the *Index*th position. If *Index* is larger than the length of *S*, no characters are deleted. If *Count* specifies more characters than remain starting at the *Index*th position, the remainder of the string is deleted.

See also *Concat, Copy, Insert, Length, Pos*

DiskFree function — WinDos

Function Returns the number of free bytes on a specified disk drive.

Declaration
```
DiskFree(Drive: Byte)
```

Result type Longint

Remarks A *Drive* of 0 indicates the default drive, 1 indicates drive *A*, 2 indicates *B*, and so on. *DiskFree* returns –1 if the drive number is invalid.

See also *DiskSize*

Example
```
uses WinDos;
begin
  Writeln(DiskFree(0) div 1024, ' Kbytes free ');
end.
```

DiskSize function — WinDos

Function Returns the total size in bytes on a specified disk drive.

Declaration
```
DiskSize(Drive: Byte)
```

Result type Longint

Remarks A *Drive* of 0 indicates the default drive, 1 indicates drive *A*, 2 indicates *B*, and so on. *DiskSize* returns –1 if the drive number is invalid.

See also *DiskFree*

Example
```
uses WinDos;
begin
  Writeln(DiskSize(0) div 1024, ' Kbytes capacity');
end.
```

Dispose procedure

Function Disposes a dynamic variable.

Declaration
```
Dispose(var P: Pointer [ , Destructor ] )
```

Remarks *P* is a pointer variable of any pointer type that was previously assigned by the *New* procedure or was assigned a meaningful value by an assignment statement. *Dispose* destroys the variable referenced by *P* and returns its

memory region to the heap. After a call to *Dispose*, the value of *P* becomes undefined, and it is an error to subsequently reference *P*^.

Dispose has been extended to allow a destructor call as a second parameter, for disposing a dynamic object type variable. In this case, *P* is a pointer variable pointing to an object type, and *Destruct* is a call to the destructor of that object type.

Restrictions If *P* does not point to a memory region in the heap, a run-time error occurs.

For a complete discussion of this topic, see "The heap manager" in Chapter 16.

See also *FreeMem, GetMem, New,*

Example

```
type
  Str18 = string[18];
var
  P: ^Str18;
begin
  New(P);
  P^ := 'Now you see it...';
  Dispose(P);                                    { Now you don't... }
end.
```

DoneWinCrt procedure — WinCrt

Function Destroys the CRT window if it hasn't already been destroyed.

Declaration `DoneWinCrt`

Remarks Calling *DoneWinCrt* just before the program ends prevents the CRT window from entering the inactive state; therefore, the user is not required to close the window.

DosVersion function — WinDos

Function Returns the DOS version number.

Declaration `DosVersion`

Result type Word

Remarks *DosVersion* returns the DOS version number. The low byte of the result is the major version number, and the high byte is the minor version number. For example, DOS 3.20 returns 3 in the low byte, and 20 in the high byte.

Example

```
uses WinDos;
var
  Ver: Word;
begin
  Ver := DosVersion;
  Writeln('This is DOS version ', Lo(Ver), '.', Hi(Ver));
end.
```

DSeg function

Function Returns the current value of the DS register.

Declaration `DSeg`

Result type Word

Remarks The result of type Word is the segment address of the data segment.

See also *CSeg*, *SSeg*

Eof function (text files)

Function Returns the end-of-file status of a text file.

Declaration `Eof [ (var F: Text) ]`

Result type Boolean

Remarks *F*, if specified, is a text-file variable. If *F* is omitted, the standard file variable *Input* is assumed. *Eof*(*F*) returns True if the current file position is beyond the last character of the file or if the file contains no components; otherwise, *Eof*(*F*) returns False.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Eoln*, *SeekEof*

Example

```
var
  F: Text;
  Ch: Char;
begin
  { Get file to read from command line }
```

```
  Assign(F, ParamStr(1));
  Reset(F);
  while not Eof(F) do
  begin
    Read(F, Ch);
    Write(Ch);                                    { Dump text file }
  end;
end.
```

Eof function (typed, untyped files)

Function Returns the end-of-file status of a typed or untyped file.

Declaration `Eof(var F)`

Result type Boolean

Remarks *F* is a file variable. *Eof(F)* returns True if the current file position is beyond the last component of the file or if the file contains no components; otherwise, *Eof(F)* returns False.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Eoln function

Function Returns the end-of-line status of a file.

Declaration `Eoln [(var F: Text)]`

Result type Boolean

Remarks *F*, if specified, is a text-file variable. If *F* is omitted, the standard file variable *Input* is assumed. *Eoln(F)* returns True if the current file position is at an end-of-line marker or if *Eof(F)* is True; otherwise, *Eoln(F)* returns False.

When checking *Eoln* on standard input that has not been redirected, the following program will wait for a carriage return to be entered before returning from the call to *Eoln*:

```
begin
{ Tells program to wait for keyboard input }
  Writeln(Eoln);
end.
```

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Eof, SeekEoln*

Erase procedure

Function Erases an external file.

Declaration
```
Erase(var F)
```

Remarks *F* is a file variable of any file type. The external file associated with *F* is erased.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions *Erase* must never be used on an open file.

See also *Rename*

Example
```
var
  F:  file;
  Ch: Char;
begin
  { Get file to delete from command line }
  Assign(F, ParamStr(1));
  {$I-}
  Reset(F);
  {$I+}
  if IOResult <> 0 then
    Writeln('Cannot find ', ParamStr(1))
  else
  begin
    Close(F);
    Write('Erase ', ParamStr(1), '? ');
    Readln(Ch);
    if UpCase(ch) = 'Y' then
      Erase(F);
  end;
end.
```

Exit procedure

Function Exits immediately from the current block.

Declaration `Exit`

Remarks When *Exit* is executed in a subroutine (procedure or function), it causes the subroutine to return. When it is executed in the statement part of a program, it causes the program to terminate. A call to *Exit* is analogous to a **goto** statement addressing a label just before the **end** of a block.

See also *Halt*

Example

```
uses WinCrt;

procedure WasteTime;
begin
  repeat
    if KeyPressed then Exit;
    Write('Xx');
  until False;
end;

begin
  WasteTime;
end.
```

Exp function

Function Returns the exponential of the argument.

Declaration `Exp(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the exponential of *X*; that is, the value *e* raised to the power of *X*, where *e* is the base of the natural logarithms.

See also *Ln*

FileExpand function — WinDos

Function Expands a file name into a fully qualified file name.

Declaration
```
FileExpand(Dest, Name: PChar)
```

Result type *PChar*

Remarks Expands the file name in *Name* into a fully qualified file name. The resulting name is converted to uppercase and consists of a drive letter, a colon, a root relative directory path, and a file name. Embedded '.' and '..' directory references are removed, and all name and extension components are truncated to 8 and 3 characters. The returned value is *Dest*. *Dest* and *Name* are allowed to refer to the same location.

Assuming that the current drive and directory is C:\SOURCE\PAS, the following *FileExpand* calls would produce these values:

```
FileExpand(S, 'test.pas')         = 'C:\SOURCE\PAS\TEST.PAS'
FileExpand(S, '..\*.TPU')         = 'C:\SOURCE\*.TPU'
FileExpand(S, 'c:\bin\turbo.exe') = 'C:\BIN\TURBO.EXE'
```

The *FileSplit* function may be used to split the result of *FileExpand* into a drive/directory string, a file-name string, and an extension string.

See also *FindFirst*, *FindNext*, *FileSplit*

FilePos function

Function Returns the current file position of a file.

Declaration
```
FilePos(var F)
```

Result type Longint

Remarks *F* is a file variable. If the current file position is at the beginning of the file, *FilePos*(*F*) returns 0. If the current file position is at the end of the file—that is, if *Eof*(*F*) is True—*FilePos*(*F*) is equal to *FileSize*(*F*).

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Cannot be used on a text file. File must be open.

See also *FileSize*, *Seek*

FileSearch function WinDos

Function Searches for a file in a list of directories.

Declaration `FileSearch(Dest, Name, List: PChar): PChar`

Result type PChar

Remarks Searches for the file given by *Name* in the list of directories given by *List*. The directories in *List* must be separated by semicolons, just like the directories specified in a PATH command in DOS. The search always starts with the current directory of the current drive. If the file is found, *FileSearch* stores a concatenation of the directory path and the file name in *Dest*. Otherwise FileSearch stores an empty string in *Dest*. The returned value is *Dest*. *Dest* and *Name* are *not* allowed to refer to the same location.

The maximum length of the result is defined by the *fsPathName* constant which is 79.

To search the PATH used by DOS to locate executable files, call *GetEnvVar*('PATH') and pass the result to *FileSearch* as the *List* parameter.

The result of *FileSearch* can be passed to *FileExpand* to convert it into a fully qualified file name, that is, an uppercase file name that includes both a drive letter and a root-relative directory path. In addition, you can use *FileSplit* to split the file name into a drive/directory string, a file-name string, and an extension string.

See also *FileExpand, FileSplit*

Example

```
uses WinCrt, WinDos;
var
  S: array[0..fsPathName] of Char;
begin
  FileSearch(S, 'TPW.EXE', GetEnvVar('PATH'));
  if S[0] = #0 then
    WriteLn('TPW.EXE not found')
  else
    WriteLn('Found as ', FileExpand(S, S));
end.
```

FileSize function

Function Returns the current size of a file.

Declaration
```
FileSize(var F)
```

Result type Longint

Remarks *F* is a file variable. *FileSize(F)* returns the number of components in *F*. If the file is empty, *FileSize(F)* returns 0.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Cannot be used on a text file. File must be open.

See also *FilePos*

Example
```
var
  F: file of Byte;
begin
  { Get file name from command line }
  Assign(F, ParamStr(1));
  Reset(F);
  Writeln('File size in bytes: ', FileSize(F));
  Close(F);
end.
```

FileSplit function WinDos

Function Splits a file name into its three components.

Declaration
```
FileSplit(Path, Dir, Name, Ext: PChar): Word
```

Remarks Splits the file name specified by *Path* into its three components. *Dir* is set to the drive and directory path with any leading and trailing backslashes, *Name* is set to the file name, and *Ext* is set to the extension with a preceding period. If a component string parameter is **nil**, the corresponding part of the path is not stored. If the path does not contain a given component, the returned component string is empty. The maximum lengths of the strings returned in Dir, Name, and Ext are defined by the *fsDirectory*, *fsFileName*, and *fsExtension* constants.

The returned value is a combination of the *fcDirectory*, *fcFileName*, and *fcExtension* bit masks, indicating which components were present in the

path. If the name or extension contains any wildcard characters (* or ?), the *fcWildcards* flag is set in the returned value.

The *fsXXXX* and *fcXXXX* constants are defined in *WinDos* as follows:

```
const
  fsPathName  = 79;
  fsDirectory = 67;
  fsFileName  = 8;
  fsExtension = 4;

const
  fcExtension = $0001;
  fcFileName  = $0002;
  fcDirectory = $0004;
  fcWildcards = $0008;
```

See also *FileExpand, FindFirst, FindNext*

Example

```
uses Strings, WinCrt, WinDos;
var
  Path: array[0..fsPathName] of Char;
  Dir: array[0..fsDirectory] of Char;
  Name: array[0..fsFileName] of Char;
  Ext: array[0..fsExtension] of Char;
begin
  Write('Filename (WORK.PAS): ');
  ReadLn(Path);
  FileSplit(Path, Dir, Name, Ext);
  if Name[0] = #0 then StrCopy(Name, 'WORK');
  if Ext[0] = #0 then StrCopy(Ext, '.PAS');
  StrECopy(StrECopy(StrECopy(Path, Dir), Name), Ext);
  WriteLn('Resulting name is ', Path);
end.
```

FillChar procedure

Function Fills a specified number of contiguous bytes with a specified value.

Declaration

```
FillChar(var X; Count: Word; Value)
```

Remarks *X* is a variable reference of any type. *Count* is an expression of type Word. *Value* is any ordinal-type expression. *FillChar* writes *Count* contiguous bytes of memory into *Value*, starting at the first byte occupied by *X*. No range-checking is performed, so be careful.

Whenever possible, use the *SizeOf* function to specify the count parameter. When using *FillChar* on strings, remember to set the length byte after the fill.

See also *Move*

Example

```
var
  S: string[80];
begin
  { Set a string to all spaces }
  FillChar(S, SizeOf(S), ' ');
  S[0] := #80;                                    { Set length byte }
end.
```


FindFirst procedure — WinDos

Function Searches the specified (or current) directory for the first entry matching the specified file name and set of attributes.

Declaration

```
FindFirst(Path: PChar; Attr: Word; var F: SearchRec)
```

Remarks *Path* is the directory mask (for example, * . *). The *Attr* parameter specifies the special files to include (in addition to all normal files). Here are the file attributes as they are declared in the *WinDos* unit:

```
const
  faReadOnly  = $01;
  faHidden    = $02;
  faSysFile   = $04;
  faVolumeID  = $08;
  faDirectory = $10;
  faArchive   = $20;
  faAnyFile   = $3F;
```

The result of the directory search is returned in the specified search record. *TSearchRec* is declared in the *WinDos* unit:

```
type
  TSearchRec = record
    Fill: array[1..21] of Byte;
    Attr: Byte;
    Time: Longint;
    Size: Longint;
    Name: array[0..12] of Char;
  end;
```

Errors are reported in *DosError*; possible error codes are 3 ("Directory Not Found") and 18 ("No More Files").

See also *FileExpand, FindNext*

Example

```
uses WinDos;

var
  DirInfo: TSearchRec;

begin
  FindFirst('*.PAS', faArchive, DirInfo);                    { Same as DIR *.PAS }
  while DosError = 0 do
  begin
    Writeln(DirInfo.Name);
    FindNext(DirInfo);
  end;
end.
```

FindNext procedure — WinDos

Function Returns the next entry that matches the name and attributes specified in a previous call to *FindFirst*.

Declaration

```
FindNext(var F: TSearchRec)
```

Remarks *S* must be the same one Passed to *FindFirst* (*TSearchRec* is declared in *WinDos* unit; see *FindFirst*). Errors are reported in *DosError*; the only possible error code is 18, which indicates no more files.

See also *FindFirst, FileExpand*

Example See the example for *FindFirst*.

Flush procedure

Function Flushes the buffer of a text file open for output.

Declaration

```
Flush(var F: Text)
```

Remarks *F* is a text-file variable.

When a text file has been opened for output using *Rewrite* or *Append*, a call to *Flush* will empty the file's buffer. This guarantees that all characters written to the file at that time have actually been written to the external file. *Flush* has no effect on files opened for input.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Frac function

Function Returns the fractional part of the argument.

Declaration `Frac(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the fractional part of *X*, that is, *Frac*(*X*) = *X* – *Int*(*X*).

See also *Int*

Example

```
var
  R: Real;
begin
  R := Frac(123.456);    { 0.456 }
  R := Frac(-123.456);   { -0.456 }
end.
```

FreeMem procedure

Function Disposes a dynamic variable of a given size.

Declaration `FreeMem(var P: Pointer; Size: Word)`

Remarks *P* is a pointer variable of any pointer type that was previously assigned by the *GetMem* procedure or was assigned a meaningful value by an assignment statement. *Size* is an expression of type Word, specifying the size in bytes of the dynamic variable to dispose; it must be *exactly* the number of bytes previously allocated to that variable by *GetMem*. *FreeMem* destroys the variable referenced by *P* and returns its memory region to the heap. If *P* does not point to a memory region in the heap, a run-time error occurs. After a call to *FreeMem*, the value of *P* becomes undefined, and it is an error to subsequently reference *P*^.

See also *Dispose, GetMem, New*

GetArgCount function — WinDos

Function Returns the number of parameters passed to the program on the command line.

Declaration
```
GetArgCount: Integer
```

GetArgStr function — WinDos

Function Returns the command-line parameter specified by *Index*.

Declaration
```
GetArgStr(Dest: PChar; Index: Integer; MaxLen: Word): PChar;
```

Remarks If *Index* is less than zero or greater than *GetArgCount*, *GetArgStr* returns an empty string. If *Index* is zero, *GetArgStr* returns the filename of the current module. *Dest* is the returned value. The maximum length of the returned string is specified by the *MaxLen* parameter.

See also *GetArgCount*

GetCBreak procedure — WinDos

Function Returns the state of *Ctrl+Break* checking in DOS.

Declaration
```
GetCBreak(var Break: Boolean)
```

Remarks *GetCBreak* returns the state of *Ctrl+Break* checking in DOS. When off (False), DOS only checks for *Ctrl+Break* during I/O to console, printer, or communication devices. When on (True), checks are made at every system call.

See also *SetCBreak*

GetCurDir function — WinDos

Function Returns the current directory of a specified drive.

Declaration
```
GetCurDir(Dir: PChar; Drive: Byte): PChar
```

Result type PChar

Remarks The string returned in *Dir* always starts with a drive letter, a colon, and a backslash. *Drive* = 0 indicates the current drive, 1 indicates drive A, 2

indicates drive B, and so on. The returned value is *Dir*. Errors are reported in *DosError*.

The maximum length of the resulting string is defined by the *fsDirectory* constant.

See also *SetCurDir, CreateDir, RemoveDir. GetDir* returns the current directory of a specified drive as a Pascal-style string.

GetDate procedure

WinDos

Function Returns the current date set in the operating system.

Declaration
```
GetDate(var Year, Month, Day, DayofWeek: Word)
```

Remarks Ranges of the values returned are *Year* 1980..2099, *Month* 1..12, *Day* 1..31, and *DayOfWeek* 0..6 (where 0 corresponds to Sunday).

See also *GetTime, SetDate, SetTime*

GetDir procedure

Function Returns the current directory of a specified drive.

Declaration
```
GetDir(D: Byte; var S: String)
```

Remarks *D* is an integer-type expression, and *S* is a string-type variable. The current directory of the drive specified by *D* is returned in *S*. *D* = 0 indicates the current drive, 1 indicates drive *A*, 2 indicates drive *B*, and so on.

GetDir performs no error-checking *per se*. If the drive specified by *D* is invalid, *S* returns '\', as if it were the root directory of the invalid drive.

See also *ChDir, DiskFree, DiskSize, MkDir, RmDir. GetCurDir* performs the same functiona as *GetDir*, but it takes a null-terminated string as an argument instead of a Pascal-style string.

GetEnvVar function — WinDos

Function Returns a pointer to the value of a specified environment variable.

Declaration
```
GetEnvVar(VarName: PChar)
```

Result type PChar

Remarks *GetEnvVar* returns a pointer to the value of a specified variable, for example, a pointer to the first character after the equals sign (=) in the environment entry given by *VarName*. The variable name can be in either uppercase or lowercase, but it must not include the equal sign (=) character. If the specified environment variable does not exist, *GetEnvVar* returns **nil**.

Example
```
uses WinCrt, WinDos;
begin
  WriteLn('The current PATH is ', GetEnvVar('PATH'));
end.
```

GetFAttr procedure — WinDos

Function Returns the attributes of a file.

Declaration
```
GetFAttr(var F; var Attr: Word);
```

Remarks *F* must be a file variable (typed, untyped, or text file) that has been assigned but not opened. The attributes are examined by **and**ing them with the file attribute masks defined as constants in the *WinDos* unit:

```
const
  faReadOnly  = $01;
  faHidden    = $02;
  faSysFile   = $04;
  faVolumeID  = $08;
  faDirectory = $10;
  faArchive   = $20;
  faAnyFile   = $3F;
```

Errors are reported in *DosError*; possible error codes are

- 3 (Invalid Path)
- 5 (File Access Denied)

Restrictions *F* cannot be open.

See also *GetFTime, SetFAttr, SetFTime*

Example

```
uses WinDos;
var
  F: file;
  Attr: Word;

begin
  { Get file name from command line }
  Assign(F, ParamStr(1));
  GetFAttr(F, Attr);
  Writeln(ParamStr(1));
  if DosError <> 0 then
    Writeln('DOS error code = ', DosError)
  else
  begin
    Write('Attribute = ', Attr);
    { Determine file attribute type using flags in WinDos unit }
    if Attr and fReadOnly <> 0 then
      Writeln('Read only file');
    if Attr and fHidden <> 0 then
      Writeln('Hidden file');
    if Attr and fSysFile <> 0 then
      Writeln('System file');
    if Attr and fVolumeID <> 0 then
      Writeln('Volume ID');
    if Attr and fDirectory <> 0 then
      Writeln('Directory name');
    if Attr and fArchive <> 0 then
      Writeln('Archive (normal file)');
  end; { else }
end.
```


GetFTime procedure — WinDos

Function Returns the date and time a file was last written.

Declaration `GetFTime(var F; var Time: Longint)`

Remarks *F* must be a file variable (typed, untyped, or text file) that has been assigned and opened. The time returned in the *Time* parameter may be unpacked through a call to *UnpackTime*. Errors are reported in *DosError*; the only possible error code is 6 (Invalid File Handle).

Restrictions *F* must be open.

See also *PackTime, SetFAttr, SetFTime, UnpackTime*

GetIntVec procedure — WinDos

Function Returns the address stored in a specified interrupt vector.

Declaration
```
GetIntVec(IntNo: Byte; var Vector: Pointer)
```

Remarks *IntNo* specifies the interrupt vector number (0..255), and the address is returned in *Vector*.

See also *SetIntVec*

GetMem procedure

Function Creates a new dynamic variable of the specified size, and puts the address of the block in a pointer variable.

Declaration
```
GetMem(var P: Pointer; Size: Word)
```

Remarks *P* is a pointer variable of any pointer type. *Size* is an expression of type Word specifying the size in bytes of the dynamic variable to allocate. The newly created variable can be referenced as *P*^.

If there isn't enough free space in the heap to allocate the new variable, a run-time error occurs. (It is possible to avoid a run-time error; see "The HeapError variable" in Chapter 16.)

Restrictions The largest block that can be allocated on the heap at one time is 65,521 bytes (64K-$F).

See also *Dispose, FreeMem, New*

GetTime procedure — WinDos

Function Returns the current time set in the operating system.

Declaration
```
GetTime(var Hour, Minute, Second, Sec100: Word)
```

Remarks Ranges of the values returned are *Hour* 0..23, *Minute* 0..59, *Second* 0..59, and *Sec*100 (hundredths of seconds) 0..99.

See also *GetDate, SetDate, SetTime, UnpackTime*

GetVerify procedure — WinDos

Function Returns the state of the verify flag in DOS.

Declaration
```
GetVerify(var Verify: Boolean)
```

Remarks *GetVerify* returns the state of the verify flag in DOS. When off (False), disk writes are not verified. When on (True), all disk writes are verified to ensure proper writing.

See also *SetVerify*

GotoXY procedure — WinCrt

Function Moves the cursor to the given coordinates within the virtual screen.

Declaration
```
GotoXY(X, Y: Integer)
```

Remarks The upper left corner corresponds to (1,1). The *Cursor* variable is set to ($X - 1$, $Y - 1$), since it stores the cursor position relative to (0, 0) instead of relative to (1, 1).

Halt procedure

Function Stops program execution and returns to the operating system.

Declaration
```
Halt [ ( ExitCode: Word ) ]
```

Remarks *ExitCode* is an optional expression of type Word that specifies the exit code of the program. *Halt* without a parameter corresponds to *Halt(0)*.

Note that *Halt* will initiate execution of any unit *Exit* procedures (see Chapter 18).

See also *Exit, RunError*

Hi function

Function Returns the high-order byte of the argument.

Declaration `Hi(X)`

Result type Byte

Remarks *X* is an expression of type Integer or Word. *Hi* returns the high-order byte of *X* as an unsigned value.

See also *Lo, Swap*

Example

```
var W: Word;
begin
  W := Hi($1234);   { $12 }
end.
```

Inc procedure

Function Increments a variable.

Declaration `Inc(var X [ ; N: Longint ] )`

Remarks *X* is an ordinal-type variable or a variable of type *PChar* if the extended syntax is enabled, and *N* is an integer-type expression. *X* is incremented by 1, or by *N* if *N* is specified; that is, *Inc*(*X*) corresponds to *X* := *X* + 1, and *Inc*(*X*, *N*) corresponds to *X* := *X* + *N*.

Inc generates optimized code and is especially useful for use in tight loops.

See also *Dec, Pred, Succ*

Example

```
var
  IntVar: Integer;
  LongintVar: Longint;
begin
  Inc(IntVar);                          { IntVar := IntVar + 1 }
  Inc(LongintVar, 5);                { LongintVar := LongintVar + 5 }
end.
```

InitWinCrt procedure — WinCrt

Function Creates the CRT window if it hasn't already been created.

Declaration
```
InitWinCrt
```

Remarks A *Read, ReadLn, Write,* or *WriteLn* with a file that has been assigned to the CRT automatically calls *InitWinCrt* to ensure that the CRT window exists. *InitWinCrt* uses the *WindowOrg, WindowSize, ScreenSize,* and *WindowTitle* variables to determine the characteristics of the CRT window.

Insert procedure

Function Inserts a substring into a string.

Declaration
```
Insert(Source: String; var S: String; Index: Integer)
```

Remarks *Source* is a string-type expression. *S* is a string-type variable of any length. *Index* is an integer-type expression. *Insert* inserts *Source* into *S* at the *Index*th position. If the resulting string is longer than 255 characters, it is truncated after the 255th character.

See also *Concat, Copy, Delete, Length, Pos*

Example
```
var
  S: String;
begin
  S := 'Honest Lincoln';
  Insert('Abe ', S, 8);                          { 'Honest Abe Lincoln' }
end.
```

Int function

Function Returns the integer part of the argument.

Declaration
```
Int(X: Real)
```

Result type Real

Remarks *X* is a real-type expression. The result is the integer part of *X*, that is, *X* rounded toward zero.

See also *Frac, Round, Trunc*

Example

```
var R: Real;
begin
  R := Int(123.456);    { 123.0 }
  R := Int(-123.456);   { -123.0 }
end.
```

Intr procedure — WinDos

Function Executes a specified software interrupt.

Declaration

```
Intr(IntNo: Byte; var Regs: TRegisters)
```

Remarks *IntNo* is the software interrupt number (0..255). *TRegisters* is a record defined in DOS:

```
type
  TRegisters = record
    case Integer of
      0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
      1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
    end;
```

Before executing the specified software interrupt, *Intr* loads the 8086 CPU's AX, BX, CX, DX, BP, SI, DI, DS, and ES registers from the *Regs* record. When the interrupt completes, the contents of the AX, BX, CX, DX, BP, SI, DI, DS, ES, and Flags registers are stored back into the *Regs* record.

To avoid general protection faults when running in Windows standard mode or Windows 386 enhanced mode, always make sure to initialize the *DS* and *ES* fields of the *TRegisters* record with valid selector values, or set the fields to zero.

Restrictions Software interrupts that depend on specific values in SP or SS on entry, or modify SP and SS on exit, cannot be executed using this procedure.

See also *MsDos*

IOResult function

Function Returns an integer value that is the status of the last I/O operation performed.

Declaration
```
IOResult
```

Result type Word

Remarks I/O-checking must be off—{**$I-**}—in order to trap I/O errors using *IOResult*. If an I/O error occurs and I/O-checking is off, all subsequent I/O operations are ignored until a call is made to *IOResult*. A call to *IOResult* clears its internal error flag.

The codes returned are summarized in Appendix A. A value of 0 reflects a successful I/O operation.

Example
```
var F: file of Byte;
begin
  { Get file name command line }
  Assign(F, ParamStr(1));
  {$I-}
  Reset(F);
  {$I+}
  if IOResult = 0 then
    Writeln('File size in bytes: ', FileSize(F))
  else
    Writeln('File not found');
end.
```

KeyPressed function — WinCrt

Function Determines if a key has been pressed on the keyboard.

Declaration
```
KeyPressed: Boolean
```

Result type Boolean

Remarks Returns *True* if a key has been pressed on the keyboard and returns *False* it no key has been pressed. The key can be read using the *ReadKey* function.

Length function

Function Returns the dynamic length of a string.

Declaration `Length(S: String)`

Result type Integer

Remarks *S* is a string-type expression. The result is the length of *S*.

See also *Concat, Copy, Delete, Insert, Pos*

Example

```
var
  F: Text;
  S: String;
begin
  Assign(F, 'GARY.PAS');
  Reset(F);
  Readln(F, S);
  Writeln('"', S, '"');
  Writeln('length = ', Length(S));
end.
```

Ln function

Function Returns the natural logarithm of the argument.

Declaration `Ln(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the natural logarithm of *X*.

See also *Exp*

Lo function

Function Returns the low-order byte of the argument.

Declaration `Lo(X)`

Result type Byte

Remarks *X* is an expression of type Integer or Word. *Lo* returns the low-order byte of *X* as an unsigned value.

See also *Hi, Swap*

Example

```
var W: Word;
begin
  W := Lo($1234);   { $34 }
end.
```

MaxAvail function

Function Returns the size of the largest contiguous free block in the heap.

Declaration `MaxAvail`

Result type Longint

Remarks *MaxAvail* compares the sizes of the largest free blocks within the heap managers sub-allocation space and the Windows global heap, and returns the larger of the two values, which corresponds to the largest dynamic variable that can be allocated at that time using *New* or *GetMem*. The size of the largest free block in the Windows global heap is calculated using the Windows *GlobalCompact* function.

See also *MemAvail*

Example

```
type
  PBuffer = ^TBuffer;
  TBuffer = array[0..16383] of Char;
var
  Buffer: PBuffer;
begin
  .
  .
  if MaxAvail < SizeOf(TBuffer) then OutOfMemory else
  begin
    New(Buffer);
    .
    .
  end;
  .
  .
end.
```

MemAvail function

Function Returns the amount of free memory in the heap.

Declaration `MemAvail`

Result type Longint

Remarks *MemAvail* calculates the amount of free memory available by calling the Windows *GetFreeSpace* function and adding to that the size of each free block in the heap managers sub-allocation space.

See also *MaxAvail*

Example

```
begin
  Writeln(MemAvail, ' bytes available');
  Writeln('Largest free block is ', MaxAvail, ' bytes');
end.
```

MkDir procedure

Function Creates a subdirectory.

Declaration `MkDir(S: String)`

Remarks *S* is a string-type expression. A new subdirectory with the path specified by *S* is created. The last item in the path cannot be an existing file name.

With **{$I-}**, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *ChDir, GetDir, RmDir. CreateDir* performs the same function as *MkDir*, but it takes a null-terminated string rather than a Pascal-style string.

Example

```
begin
  {$I-}
  { Get directory name from command line }
  MkDir(ParamStr(1));
  if IOResult <> 0 then
    Writeln('Cannot create directory')
  else
    Writeln('New directory created');
end.
```

Move procedure

Function Copies a specified number of contiguous bytes from a source range to a destination range.

Declaration
```
Move(var Source, Dest; Count: Word)
```

Remarks *Source* and *Dest* are variable references of any type. *Count* is an expression of type Word. *Move* copies a block of *Count* bytes from the first byte occupied by *Source* to the first byte occupied by *Dest*. No checking is performed, so be careful with this procedure.

When *Source* and *Dest* are in the same segment, that is, when the segment parts of their addresses are equal, *Move* automatically detects and compensates for any overlap. Intrasegment overlaps never occur on statically and dynamically allocated variables (unless they are deliberately forced), and they are therefore not detected.

Whenever possible, use the *SizeOf* function to determine the *Count*.

See also *FillChar*

Example
```
var
  A: array[1..4] of Char;
  B: Longint;
begin
  Move(A, B, SizeOf(A));                    { SizeOf = safety! }
end.
```

MsDos procedure — WinDos

Function Executes a DOS function call.

Declaration
```
MsDos(var Regs: TRegisters)
```

Remarks The effect of a call to *MsDos* is the same as a call to *Intr* with an *IntNo* of $21. *TRegisters* is a record declared in the *WinDos* unit:

```
type
  TRegisters = record
    case Integer of
      0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
      1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
    end;
```

Restrictions Software interrupts that depend on specific calls in SP or SS on entry or modify SP and SS on exit cannot be executed using this procedure.

See also *Intr*

New procedure

Function Creates a new dynamic variable and sets a pointer variable to point to it.

Declaration
```
New(var P: Pointer [ , Init: Constructor ] )
```

Remarks *P* is a pointer variable of any pointer type. The size of the allocated memory block corresponds to the size of the type that *P* points to. The newly created variable can be referenced as *P*^. If there isn't enough free space in the heap to allocate the new variable, a run-time error occurs. (It is possible to avoid a run-time error in this case; see "The HeapError variable" in Chapter 16.)

New has been extended to allow a constructor call as a second parameter for allocating a dynamic object type variable. *P* is a pointer variable, pointing to an object type, and *Construct* is a call to the constructor of that object type.

An additional extension allows *New* to be used as a *function*, which allocates and returns a dynamic variable of a specified type. If the call is of the form *New(P)*, *P* can be any pointer type. If the call is of the form *New(P, Init)*, *P* must point to an object type, and *Init* must be a call to the constructor of that object type. In both cases, the type of the function result is *P*.

See also *Dispose, FreeMem, GetMem*

Odd function

Function Tests if the argument is an odd number.

Declaration
```
Odd(X: Longint)
```

Result type Boolean

Remarks *X* is a Longint-type expression. The result is True if *X* is an odd number, and False if *X* is an even number.

Ofs function

Function Returns the offset of a specified object.

Declaration `Ofs(X)`

Result type Word

Remarks *X* is any variable, or a procedure or function identifier. The result of type Word is the offset part of the address of *X*.

See also *Addr, Seg*

Ord function

Function Returns the ordinal number of an ordinal-type value.

Declaration `Ord(X)`

Result type Longint

Remarks *X* is an ordinal-type expression. The result is of type Longint and its value is the ordinality of *X*.

See also *Chr*

PackTime procedure — WinDos

Function Converts a *TDateTime* record into a 4-byte, packed date-and-time Longint used by *SetFTime*.

Declaration `PackTime(var DT: TDateTime; var Time: Longint)`

Remarks *TDateTime* is a record declared in the *WinDos* unit:

```
TDateTime = record
  Year, Month, Day, Hour, Min, Sec: Word;
end;
```

The fields of the *TDateTime* record are not range-checked.

See also *GetFTime, GetTime, SetFTime, SetTime, UnpackTime*

ParamCount function

Function Returns the number of parameters passed to the program on the command line.

Declaration `ParamCount`

Result type Word

Remarks Blanks and tabs serve as separators.

See also *ParamStr*

Example

```
begin
  if ParamCount < 1 then
    Writeln('No parameters on command line')
  else
    Writeln(ParamCount, ' parameter(s)');
end.
```

ParamStr function

Function Returns a specified command-line parameter.

Declaration `ParamStr(Index)`

Result type String

Remarks *Index* is an expression of type Word. *ParamStr* returns the *Index*th parameter from the command line, or an empty string if *Index* is zero or greater than *ParamCount*. With DOS 3.0 or later, *ParamStr(0)* returns the path and file name of the executing program (for example, C:\TP\MYPROG.EXE).

See also *ParamCount*

Example

```
var I: Word;
begin
  for I := 1 to ParamCount do
    Writeln(ParamStr(I));
end.
```

Pi function

Function Returns the value of Pi (3.1415926535897932385).

Declaration `Pi`

Result type Real

Remarks Precision varies, depending on whether the compiler is in 80x87 or software-only mode.

Pos function

Function Searches for a substring in a string.

Declaration `Pos(Substr, S: String)`

Result type Byte

Remarks *Substr* and *S* are string-type expressions. *Pos* searches for *Substr* within *S*, and returns an integer value that is the index of the first character of *Substr* within *S*. If *Substr* is not found, *Pos* returns zero.

See also *Concat, Copy, Delete, Insert, Length*

Example

```
var S: String;
begin
  S := '   123.5';
  { Convert spaces to zeroes }
  while Pos(' ', S) > 0 do
    S[Pos(' ', S)] := '0';
end.
```

Pred function

Function Returns the predecessor of the argument.

Declaration `Pred(X)`

Result type Same type as parameter.

Remarks *X* is an ordinal-type expression. The result, of the same type as *X*, is the predecessor of *X*.

See also *Dec, Inc, Succ*

Ptr function

Function Converts a segment base and an offset address to a pointer-type value.

Declaration
```
Ptr(Seg, Ofs: Word)
```

Result type Pointer

Remarks *Seg* and *Ofs* are expressions of type Word. The result is a pointer that points to the address given by *Seg* and *Ofs*. Like **nil**, the result of *Ptr* is assignment-compatible with all pointer types.

The function result may be dereferenced and typecast:

```
if Byte(Ptr($40, $49)^) = 7 then
  Writeln('Video mode = mono');
```

See also *Addr, Ofs, Seg*

Example
```
var P: ^Byte;
begin
  P := Ptr($40, $49);
  Writeln('Current video mode is ', P^);
end.
```

Random function

Function Returns a random number.

Declaration
```
Random [ ( Range: Word) ]
```

Result type Real or Word, depending on the parameter

Remarks If *Range* is not specified, the result is a *Real* random number within the range 0 <= *X* < 1. If *Range* is specified, it must be an expression of type Integer, and the result is a Word random number within the range 0 <= *X* < *Range*. If *Range* equals 0, a value of 0 will be returned.

The *Random* number generator should be initialized by making a call to *Randomize*, or by assigning a value to *RandSeed*.

See also *Randomize*

Randomize procedure

Function Initializes the built-in random generator with a random value.

Declaration `Randomize`

Remarks The random value is obtained from the system clock.

The random-number generator's seed is stored in a predeclared Longint variable called *RandSeed*. By assigning a specific value to *RandSeed*, a specific sequence of random numbers can be generated over and over. This is particularly useful in applications that use data encryption.

See also *Random*

ReadBuf function WinCrt

Function Reads a line from the CRT window.

Declaration `ReadBuf(Buffer: PChar; Count: Word)`

Result type Word

Remarks *Buffer* points to a line buffer that has room for up to *Count* characters. Up to *Count* – 2 characters can be entered, and an end-of-line marker (a #13 followed by a #10) is automatically appended to the line when the user presses *Enter*. If *CheckEOF* is *True*, the user can also terminate the input line by pressing *Ctrl+Z* and the line will have an end-of-file marker (#26) appended to it. The return value is the number of characters read, including the end-of-file marker.

See also *ReadChar*

Read procedure (text files)

Function Reads one or more values from a text file into one or more variables.

Declaration `Read( [ var F: Text; ] V1 [, V2,...,VN ] )`

Remarks F, if specified, is a text-file variable. If *F* is omitted, the standard file variable *Input* is assumed. Each *V* is a variable of type Char, Integer, Real, or String.

With a type Char variable, *Read* reads one character from the file and assigns that character to the variable. If *Eof(F)* was True before *Read* was executed, the value *Chr*(26) (a *Ctrl+Z* character) is assigned to the variable. If *Eoln(F)* was True, the value *Chr(13)* (a carriage-return character) is assigned to the variable. The next *Read* will start with the next character in the file.

With a type integer variable, *Read* expects a sequence of characters that form a signed number, according to the syntax shown in the section "Numbers" in Chapter 1. Any blanks, tabs, or end-of-line markers preceding the numeric string are skipped. Reading ceases at the first blank, tab, or end-of-line marker following the numeric string or if *Eof(F)* becomes True. If the numeric string does not conform to the expected format, an I/O error occurs; otherwise, the value is assigned to the variable. If *Eof(F)* was True before *Read* was executed or if *Eof(F)* becomes True while skipping initial blanks, tabs, and end-of-line markers, the value 0 is assigned to the variable. The next *Read* will start with the blank, tab, or end-of-line marker that terminated the numeric string.

With a type real variable, *Read* expects a sequence of characters that form a signed whole number, according to the syntax shown in the section "Numbers" in Chapter 1 (except that hexadecimal notation is not allowed). Any blanks, tabs, or end-of-line markers preceding the numeric string are skipped. Reading ceases at the first blank, tab, or end-of-line marker following the numeric string or if *Eof(F)* becomes True. If the numeric string does not conform to the expected format, an I/O error occurs; otherwise, the value is assigned to the variable. If *Eof(F)* was True before *Read* was executed, or if *Eof(F)* becomes True while skipping initial blanks, tabs, and end-of-line markers, the value 0 is assigned to the variable. The next *Read* will start with the blank, tab, or end-of-line marker that terminated the numeric string.

With a type string variable, *Read* reads all characters up to, but not including, the next end-of-line marker or until *Eof(F)* becomes True. The resulting character string is assigned to the variable. If the resulting string is longer than the maximum length of the string variable, it is truncated. The next *Read* will start with the end-of-line marker that terminated the string.

When the extended syntax is enabled, *Read* can also be used to read null-terminated strings into zero-based character arrays. With a character array of the form **array**[0..N] **of** Char, *Read* reads up to *N* characters, or until *Eoln(F)* or *Eof(F)* become *True*, and then appends a NULL (#0) terminator to the string.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions *Read* with a type string variable does not skip to the next line after reading. For this reason, you cannot use successive *Read* calls to read a sequence of strings, since you will never get past the first line; after the first *Read*, each subsequent *Read* will see the end-of-line marker and return a zero-length string. Instead, use multiple *Readln* calls to read successive string values.

See also *Readln, ReadKey, Write, Writeln*

Read procedure (typed files)

Function Reads a file component into a variable.

Declaration
```
Read(F, V1 [, V2,...,VN ] )
```

Remarks *F* is a file variable of any type except text, and each *V* is a variable of the same type as the component type of *F*. For each variable read, the current file position is advanced to the next component. It's an error to attempt to read from a file when the current file position is at the end of the file, that is, when *Eof(F)* is True.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions File must be open.

See also *Write*

ReadKey function WinCrt

Function Reads a character from the keyboard.

Declaration
```
ReadKey
```

Result type Char

Remarks The character is *not* echoed to the screen. The *ReadKey* function only supports standard ASCII key codes. It does not support extended key codes, such as function and cursor key codes.

ReadIn procedure

Function Executes the *Read* procedure then skips to the next line of the file.

Declaration
```
Readln( [ var F: Text; ] V1 [, V2,...,VN ] )
```

Remarks *Readln* is an extension to *Read*, as it is defined on text files. After executing the *Read*, *Readln* skips to the beginning of the next line of the file.

Readln(F) with no parameters causes the current file position to advance to the beginning of the next line (if there is one; otherwise, it goes to the end of the file). *Readln* with no parameter list altogether corresponds to *Readln(Input)*.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Works only on text files, including standard input. File must be open for input.

See also *Read*

RemoveDir procedure WinDos

Function Removes an empty subdirectory.

Declaration
```
Remove(Dir: PChar)
```

Remarks The subdirectory with the path specified by *Dir* is removed. Errors, such as a non-existing or non-empty subdirectory, are reported in the *DosError* variable.

See also *GetCurDir*, *CreateDir*, *SetCurDir*. *RmDir* removes an empty subdirectory also, but it takes a Pascal-style string as the argument rather than a null-terminated string.

Rename procedure

Function Renames an external file.

Declaration
```
Rename(var F; Newname)
```

Remarks *F* is a file variable of any file type. *Newname* is a string-type expression or an expression of type *PChar* if the extended syntax is enabled. The

external file associated with *F* is renamed to *Newname*. Further operations on *F* will operate on the external file with the new name.

With {**$I-**}, *IOResult* returns 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions *Rename* must never be used on an open file.

See also *Erase*

Reset procedure

Function Opens an existing file.

Declaration
```
Reset(var F [: file; RecSize: Word ] )
```

Remarks *F* is a file variable of any file type, which must have been associated with an external file using *Assign*. *RecSize* is an optional expression of type Word, which can only be specified if *F* is an untyped file.

Reset opens the existing external file with the name assigned to *F*. It's an error if no existing external file of the given name exists. If *F* was already open, it is first closed and then re-opened. The current file position is set to the beginning of the file.

If *F* was assigned an empty name, such as *Assign*(*F*, ''), then after the call to *Reset*, *F* will refer to the standard input file (standard handle number 0).

If *F* is a text file, *F* becomes read-only. After a call to *Reset*, *Eof*(*F*) is True if the file is empty; otherwise, *Eof*(*F*) is False.

If *F* is an untyped file, *RecSize* specifies the record size to be used in data transfers. If *RecSize* is omitted, a default record size of 128 bytes is assumed.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Append, Assign, Close, Rewrite, Truncate*

Example
```
function FileExists(FileName: String): Boolean;
{ Boolean function that returns True if the file exists; otherwise, it returns
  False. Closes the file if it exists. }
var
  F: file;
begin
  {$I-}
  Assign(F, FileName);
```

```
  Reset(F);
  Close(F);
  {$I+}
  FileExists := (IOResult = 0) and (FileName <> '');
end;  { FileExists }

begin
  if FileExists(ParamStr(1)) then              { Get file name from command line }
    Writeln('File exists')
  else
    Writeln('File not found');
end.
```

Rewrite procedure

Function Creates and opens a new file.

Declaration

```
Rewrite(var F [: file; RecSize: Word ] )
```

Remarks *F* is a file variable of any file type, which must have been associated with an external file using *Assign*. *RecSize* is an optional expression of type Word, which can only be specified if *F* is an untyped file.

Rewrite creates a new external file with the name assigned to *F*. If an external file with the same name already exists, it is deleted and a new empty file is created in its place. If *F* was already open, it is first closed and then re-created. The current file position is set to the beginning of the empty file.

If *F* was assigned an empty name, such as *Assign*(*F*, ''), then after the call to *Rewrite*, *F* will refer to the standard output file (standard handle number 1).

If *F* is a text file, *F* becomes write-only. After a call to *Rewrite*, *Eof*(*F*) is always True.

If *F* is an untyped file, *RecSize* specifies the record size to be used in data transfers. If *RecSize* is omitted, a default record size of 128 bytes is assumed.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Append*, *Assign*, *Reset*, *Truncate*

Example

```
var F: Text;
begin
  Assign(F, 'NEWFILE.$$$');
```

```
  Rewrite(F);
  Writeln(F, 'Just created file with this text in it...');
  Close(F);
end.
```

RmDir procedure

Function Removes an empty subdirectory.

Declaration
```
RmDir(S: String)
```

Remarks *S* is a string-type expression. The subdirectory with the path specified by *S* is removed. If the path does not exist, is non-empty, or is the currently logged directory, an I/O error will occur.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *MkDir*, *ChDir*, *GetDir*. *RemoveDir* performs the same function as *RmDir*, but it takes a null-terminated string as an arugment rather than a Pascal-style string.

Example
```
begin
  {$I-}
  { Get directory name from command line }
  RmDir(ParamStr(1));
  if IOResult <> 0 then
    Writeln('Cannot remove directory')
  else
    Writeln('directory removed');
end.
```

Round function

Function Rounds a real-type value to an integer-type value.

Declaration
```
Round(X: Real)
```

Result type Longint

Remarks *X* is a real-type expression. *Round* returns a Longint value that is the value of *X* rounded to the nearest whole number. If *X* is exactly halfway between two whole numbers, the result is the number with the greatest absolute magnitude. A run-time error occurs if the rounded value of *X* is not within the Longint range.

See also *Int, Trunc*

RunError procedure

Function Stops program execution and generates a run-time error.

Declaration
```
RunError [ ( ErrorCode: Byte ) ]
```

Remarks The *RunError* procedure corresponds to the *Halt* procedure except that in addition to stopping the program, it generates a run-time error at the current statement. *ErrorCode* is the run-time error number (0 if omitted). If the current module is compiled with **D**ebug Information checked (turned on), and you're running the program from the IDE, Turbo Pascal automatically takes you to the *RunError* call, just as if an ordinary run-time error had occurred.

See also *Exit, Halt*

Example
```
{$IFDEF Debug}
  if P = nil then
    RunError(204);
{$ENDIF}
```

ScrollTo procedure WinCrt

Function Scrolls the CRT window to show the virtual screen location given by (X, Y) in the upper left corner.

Declaration
```
ScrollTo(X, Y: Integer)
```

Remarks The location described with coordinates (0,0) corresponds to the upper left corner of the virtual screen.

Seek procedure

Function Moves the current position of a file to a specified component.

Declaration
```
Seek(var F; N: Longint)
```

Remarks *F* is any file variable type except text, and *N* is an expression of type Longint. The current file position of *F* is moved to component number *N*. The number of the first component of a file is 0. In order to expand a file, it

is possible to seek one component beyond the last component; that is, the statement *Seek(F, FileSize(F))* moves the current file position to the end of the file.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Cannot be used on text files. File must be open.

See also *FilePos*

SeekEof function

Function Returns the end-of-file status of a file.

Declaration
```
SeekEof [ (var F: Text) ]
```

Result type Boolean

Remarks *SeekEof* corresponds to *Eof* except that it skips all blanks, tabs, and end-of-line markers before returning the end-of-file status. This is useful when reading numeric values from a text file.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Can only be used on text files. File must be open.

See also *Eof, SeekEoln*

SeekEoln function

Function Returns the end-of-line status of a file.

Declaration
```
SeekEoln [ (var F: Text) ]
```

Result type Boolean

Remarks *SeekEoln* corresponds to *Eoln* except that it skips all blanks and tabs before returning the end-of-line status. This is useful when reading numeric values from a text file.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions Can only be used on text files. File must be open.

See also *Eoln, SeekEof*

Seg function

Function Returns the segment of a specified object.

Declaration `Seg(X)`

Result type Word

Remarks *X* is any variable, or a procedure or function identifier. The result, of type Word, is the segment part of the address of *X*.

See also *Addr*, *Ofs*

SetCBreak procedure — WinDos

Function Sets the state of *Ctrl+Break* checking in DOS.

Declaration `SetCBreak(Break: Boolean)`

Remarks *SetCBreak* sets the state of *Ctrl+Break* checking in DOS. When off (False), DOS only checks for *Ctrl+Break* during I/O to console, printer, or communication devices. When on (True), checks are made at every system call.

See also *GetCBreak*

SetCurDir procedure — WinDos

Function Changes the current directory to the path specified by *Dir*.

Declaration `SetCurDir(Dir: PChar)`

Remarks If *Dir* specified a drive letter, the current drive is also changed. Errors are reported in *DosError*.

See also *GetCurDir*, *CreateDir*, *RemoveDir*. *ChDir* performs the same function as *SetCurDir*, but it takes a Pascal-style string as the argument rather than a null-terminated string.

SetDate procedure — WinDos

Function Sets the current date in the operating system.

Declaration
```
SetDate(Year, Month, Day: Word)
```

Remarks Valid parameter ranges are *Year* 1980..2099, *Month* 1..12, and *Day* 1..31. If the date is invalid, the request is ignored.

See also *GetDate, GetTime, SetTime*

SetFAttr procedure — WinDos

Function Sets the attributes of a file.

Declaration
```
SetFAttr(var F; Attr: Word)
```

Remarks *F* must be a file variable (typed, untyped, or text file) that has been assigned but not opened. The attribute value is formed by adding the appropriate attribute masks defined as constants in the *WinDos* unit.

```
const
  faReadOnly  = $01;
  faHidden    = $02;
  faSysFile   = $04;
  faVolumeID  = $08;
  faDirectory = $10;
  faArchive   = $20;
```

Errors are reported in *DosError*; possible error codes are 3 (Invalid Path) and 5 (File Access Denied).

Restrictions *F* cannot be open.

See also *GetFAttr, GetFTime, SetFTime*

Example
```
uses WinDos;
var
  F: file;
begin
  Assign(F, 'C:\AUTOEXEC.BAT');
  SetFAttr(F, faHidden);                                    { Uh-oh }
  Readln;
  SetFAttr(F, faArchive);                                   { Whew! }
end.
```

SetFTime procedure — WinDos

Function Sets the date and time a file was last written.

Declaration
```
SetFTime(var F; Time: Longint)
```

Remarks *F* must be a file variable (typed, untyped, or text file) that has been assigned and opened. The *Time* parameter can be created through a call to *PackTime*. Errors are reported in *DosError*; the only possible error code is 6 (Invalid File Handle).

Restrictions *F* must be open.

See also *GetFTime, PackTime, SetFAttr, UnpackTime*

SetIntVec procedure — WinDos

Function Sets a specified interrupt vector to a specified address.

Declaration
```
SetIntVec(IntNo: Byte; Vector: Pointer)
```

Remarks *IntNo* specifies the interrupt vector number (0..255), and *Vector* specifies the address. *Vector* is often constructed with the @ operator to produce the address of an interrupt procedure. Assuming *Int1BSave* is a variable of type Pointer, and *Int1BHandler* is an interrupt procedure identifier, the following statement sequence installs a new interrupt $1*B* handler and later restores the original handler:

```
GetIntVec($1B, Int1BSave);
SetIntVec($1B, @Int1BHandler);
...
SetIntVec($1B, Int1BSave);
```

See also *GetIntVec*

SetTextBuf procedure

Function Assigns an I/O buffer to a text file.

Declaration
```
SetTextBuf(var F: Text; var Buf [ ; Size: Word ] )
```

Remarks *F* is a text-file variable, *Buf* is any variable, and *Size* is an optional expression of type Word.

Each text-file variable has an internal 128-byte buffer that, by default, is used to buffer *Read* and *Write* operations. This buffer is adequate for most applications. However, heavily I/O-bound programs, such as applications that copy or convert text files, will benefit from a larger buffer, because it reduces disk head movement and file system overhead.

SetTextBuf changes the text file *F* to use the buffer specified by *Buf* instead of *F*'s internal buffer. *Size* specifies the size of the buffer in bytes. If *Size* is omitted, *SizeOf(Buf)* is assumed; that is, by default, the entire memory region occupied by *Buf* is used as a buffer. The new buffer remains in effect until *F* is next passed to *Assign*.

Restrictions

SetTextBuf should never be applied to an open file, although it can be called immediately after *Reset*, *Rewrite*, and *Append*. Calling *SetTextBuf* on an open file once I/O operations has taken place can cause loss of data because of the change of buffer.

Turbo Pascal doesn't ensure that the buffer exists for the entire duration of I/O operations on the file. In particular, a common error is to install a local variable as a buffer, and then use the file outside the procedure that declared the buffer.

Example

```
var
  F: Text;
  Ch: Char;
  Buf: array[1..10240] of Char;                    { 10K buffer }
begin
  { Get file to read from command line }
  Assign(F, ParamStr(1));
  { Bigger buffer for faster reads }
  SetTextBuf(F, Buf);
  Reset(F);
  { Dump text file onto screen }
  while not Eof(f) do
  begin
    Read(F, Ch);
    Write(Ch);
  end;
end.
```

SetTime procedure — WinDos

Function Sets the current time in the operating system.

Declaration
```
SetTime(Hour, Minute, Second, Sec100: Word)
```

Remarks Valid parameter ranges are *Hour* 0..23, *Minute* 0..59, *Second* 0..59, and *Sec*100 (hundredths of seconds) 0..99. If the time is not valid, the request is ignored.

See also *GetDate, GetTime, PackTime, SetDate, UnpackTime*

SetVerify procedure — WinDos

Function Sets the state of the verify flag in DOS.

Declaration
```
SetVerify(Verify: Boolean)
```

Remarks *SetVerify* sets the state of the verify flag in DOS. When off (False), disk writes are not verified. When on (True), all disk writes are verified to ensure proper writing.

See also *GetVerify*

Sin function

Function Returns the sine of the argument.

Declaration
```
Sin(X: Real)
```

Result type Real

Remarks *X* is a real-type expression. The result is the sine of *X*. *X* is assumed to represent an angle in radians.

See also *ArcTan, Cos*

Example
```
var
  R: Real;
begin
  R := Sin(Pi);
end.
```

SizeOf function

Function Returns the number of bytes occupied by the argument.

Declaration `SizeOf(X)`

Result type Word

Remarks *X* is either a variable reference or a type identifier. *SizeOf* returns the number of bytes of memory occupied by *X*.

SizeOf should always be used when passing values to *FillChar*, *Move*, *GetMem*, and so on:

```
FillChar(S, SizeOf(S), 0);
GetMem(P, SizeOf(RecordType));
```

Example

```
type
  CustRec = record
    Name: string[30];
    Phone: string[14];
  end;
var
  P: ^CustRec;
begin
  GetMem(P, SizeOf(CustRec));
end.
```

SPtr function

Function Returns the current value of the SP register.

Declaration `SPtr`

Result type Word

Remarks The result, of type Word, is the offset of the stack pointer within the stack segment.

See also *SSeg*

Sqr function

Function Returns the square of the argument.

Declaration `Sqr(X)`

Result type Same type as parameter.

Remarks *X* is an integer-type or real-type expression. The result, of the same type as *X*, is the square of *X*, or *X* * *X*.

Sqrt function

Function Returns the square root of the argument.

Declaration `Sqrt(X: Real)`

Result type Real

Remarks *X* is a real-type expression. The result is the square root of *X*.

SSeg function

Function Returns the current value of the SS register.

Declaration `SSeg`

Result type Word

Remarks The result, of type Word, is the segment address of the stack segment.

See also *SPtr*, *CSeg*, *DSeg*

Str procedure

Function Converts a numeric value to its string representation.

Declaration `Str(X [: Width [: Decimals ] ]; var S)`

Remarks *X* is an integer-type or real-type expression. *Width* and *Decimals* are integer-type expressions. *S* is a string-type variable or a zero-based character array variable if the extended syntax is enabled. *Str* converts *X* to its string representation, according to the *Width* and *Decimals* formatting parameters. The effect is exactly the same as a call to the *Write*

standard procedure with the same parameters, except that the resulting string is stored in *S* instead of being written to a text file.

See also *Val, Write*

Example

```
function IntToStr(I: Longint): String;
{ Convert any integer type to a string }
var
  S: string[11];
begin
  Str(I, S);
  IntToStr := S;
end;
begin
  Writeln(IntToStr(-5322));
end.
```

StrCat function — Strings

Function Appends a copy of one string to the end of another and returns the concatenated string.

Declaration

```
StrCat(Dest, Source: PChar)
```

Result type PChar

Remarks *StrCat* appends a copy of *Source* to *Dest* and returns *Dest*. *StrCat* does not perform any length checking. It is up to you to ensure that the buffer given by *Dest* has room for at least *StrLen(Dest)* + *StrLen(Source)* + 1 characters. If you want length checking, use the *StrLCat* function.

See also *StrLCat*

Example

```
uses Strings, WinCrt;
const
  Turbo: PChar = 'Turbo';
  Pascal: PChar = 'Pascal';
var
  S: array[0..15] of Char;
begin
  StrCopy(S, Turbo);
  StrCat(S, ' ');
  StrCat(S, Pascal);
  WriteLn(S);
end.
```

StrComp function — Strings

Function Compares two strings.

Declaration `StrComp(Str1, Str2: PChar)`

Result type Integer

Remarks *StrComp* compares *Str1* to *Str2*. The return value is less than 0 if *Str1* < *Str2*, 0 if *Str1* = *Str2*, or greater than 0 if *Str1* > *Str2*.

See also *StrIComp, StrLComp, StrLIComp*

Example

```
uses Strings, WinCrt;
var
  C: Integer;
  Result: PChar;
  S1, S2: array[0..79] of Char;
begin
  ReadLn(S1);
  ReadLn(S2);
  C := StrComp(S1, S2);
  if C < 0 then Result := ' is less than ' else
    if C > 0 then Result := ' is greater than ' else
      Result := ' is equal to ';
  WriteLn(S1, Result, S2);
end.
```

StrCopy function — Strings

Function Copies one string to another.

Declaration `StrCopy(Dest, Source: PChar)`

Result type PChar

Remarks *StrCopy* copies *Source* to *Dest* and returns *Dest*. *StrCopy* does not perform any length checking. It is up to you to ensure that the buffer given by *Dest* has room for at least *StrLen(Source)* + 1 characters. If you want length checking, use the *StrLCopy* function.

See also *StrECopy, StrLCopy*

Example

```
uses Strings, WinCrt;
var
  S: array[0..15] of Char;
begin
```

```
  StrCopy(S, 'Turbo Pascal');
  WriteLn(S);
end.
```

StrDispose function — Strings

Function Disposes a string on the heap.

Declaration `StrDispose(Str: PChar)`

Remarks StrDispose disposes of a string that was previously allocated with *StrNew*. If *Str* is **nil**, *StrDispose* does nothing.

See also StrNew

StrECopy function — Strings

Function Copies one string to another, returning a pointer to the end of the resulting string.

Declaration `StrECopy(Dest, Source: PChar)`

Result type PChar

Remarks *StrECopy* copies *Source* to *Dest* and returns *StrEnd(Dest)*. It is up to you to ensure that the buffer given by *Dest* has room for at least *StrLen(Source)* + 1 characters. Nested calls to *StrECopy* can be used to concatenate a sequence of strings; this is illustrated by the example that follows.

See also *StrCopy, StrEnd*

Example

```
uses Strings, WinCrt;
const
  Turbo: PChar = 'Turbo';
  Pascal: PChar = 'Pascal';
var
  S: array[0..15] of Char;
begin
  StrECopy(StrECopy(StrECopy(S, Turbo), ' '), Pascal);
  WriteLn(S);
end.
```

StrEnd function — Strings

Function Returns a pointer to the end of a string.

Declaration `StrEnd(Str: PChar)`

Result type PChar

Remarks *StrEnd* returns a pointer to the null character that terminates *Str*.

See also *StrLen*

Example

```
uses Strings, WinCrt;
var
  S: array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn('String length is ', StrEnd(S) - S);
end.
```

StrIComp function — Strings

Function Compares two strings without case sensitivity.

Declaration `StrIComp(Str1, Str2: PChar)`

Result type Integer

Remarks *StrIComp* compares *Str1* to *Str2* without case sensitivity. The return value is the same as *StrComp*.

See also *StrComp, StrLComp, StrLIComp*

StrLCat function — Strings

Function Appends characters from a string to the end of another, and returns the concatenated string.

Declaration `StrLCat(Dest, Source: PChar; MaxLen: Word)`

Result type PChar

Remarks *StrLCat* appends at most *MaxLen – StrLen(Dest)* characters from *Source* to the end of *Dest*, and returns *Dest*. The *SizeOf* standard function can be used to determine the *MaxLen* parameter; this is demonstrated by the example that follows.

See also *StrCat*

Example

```
uses Strings, WinCrt;
var
  S: array[0..9] of Char;
begin
  StrLCopy(S, 'Turbo', SizeOf(S) - 1)
  StrLCat(S, ' ', SizeOf(S) - 1);
  StrLCat(S, 'Pascal', SizeOf(S) - 1);
  WriteLn(S);
end.
```

StrLComp function — Strings

Function Compares two strings, up to a maximum length.

Declaration

```
StrLComp(Str1, Str2: PChar; MaxLen: Word)
```

Result type Integer

Remarks *StrLComp* compares *Str1* to *Str2*, up to a maximum length of *MaxLen* characters. The return value is the same as *StrComp*.

See also *StrComp, StrLIComp, StrIComp*

Example

```
uses Strings, WinCrt;
var
  Result: PChar;
  S1, S2: array[0..79] of Char;
begin
  ReadLn(S1);
  ReadLn(S2);
  if StrLComp(S1, S2, 5) = 0 then
    Result := 'equal'
  else
    Result := 'different';
  WriteLn('The first five characters are ', Test);
end.
```

StrLCopy function — Strings

Function Copies characters from one string to another.

Declaration `StrLCopy(Dest, Source: PChar; MaxLen: Word)`

Result type PChar

Remarks *StrLCopy* copies at most *MaxLen* characters from *Source* to *Dest* and returns *Dest*. The *SizeOf* standard function can be used to determine the *MaxLen* parameter; this is demonstrated by the example that follows.

See also *StrCopy*

Example

```
uses Strings, WinCrt;
var
  S: array[0..9] of Char;
begin
  StrLCopy(S, 'Turbo Pascal', SizeOf(S) - 1);
  WriteLn(S);
end.
```

StrLen function — Strings

Function Returns the number of characters in *Str*.

Declaration `StrLen(Str: PChar)`

Result type Word

Remarks *StrLen* returns the number of characters in *Str*, not counting the null terminator.

See also *StrEnd*

Example

```
uses Strings, WinCrt;
var
  S: array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn('String length is ', StrLen(S));
end.
```

StrLIComp function — Strings

Function Compares two strings, up to a maximum length, without case sensitivity.

Declaration
```
StrLIComp(Str1, Str2: PChar; MaxLen: Word)
```

Result type Integer

Remarks *StrLIComp* compares *Str1* to *Str2*, up to a maximum length of *MaxLen* characters, without case sensitivity. The return value is the same as *StrComp*.

See also *StrComp*, *StrIComp*, *StrLComp*

StrLower function — Strings

Function Converts a string to lowercase.

Declaration
```
StrLower(Str: PChar)
```

Result type PChar

Remarks *StrLower* converts *Str* to lowercase and returns *Str*.

See also *StrUpper*

Example
```
uses Strings, WinCrt;
var
  S: array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn(StrLower(S));
  WriteLn(StrUpper(S));
end.
```

StrMove function — Strings

Function Copies characters from one string to another.

Declaration `StrMove(Dest, Source: PChar; Count: Word)`

Result type PChar

Remarks *StrMove* copies exactly *Count* characters from *Source* to *Dest* and returns *Dest*. *Source* and *Dest* may overlap.

Example

```
{ Allocate string on heap }

function StrNew(S: PChar): PChar;
var
  L: Word;
  P: PChar;
begin
  if (S = nil) or (S^ = #0) then StrNew := nil else
  begin
    L := StrLen(S) + 1;
    GetMem(P, L);
    StrNew := StrMove(P, S, L);
  end;
end;

{ Dispose string on heap }

procedure StrDispose(S: PChar);
begin
  if S <> nil then FreeMem(S, StrLen(S) + 1);
end;
```

StrNew function — Strings

Function Allocates a string on the heap.

Declaration `StrNew(Str: PChar)`

Result type PChar

Remarks *StrNew* allocates a copy of Str on the heap. If *Str* is **nil** or points to an empty string, *StrNew* returns **nil** and doesn't allocate any heap space. Otherwise, *StrNew* makes a duplicate of *Str*, obtaining space with a call to the *GetMem* standard procedure, and returns a pointer to the duplicated string. The allocated space is *StrLen(Str)* + 1 bytes long.

See also *StrDispose*

Example

```
uses Strings, WinCrt;
var
  P: PChar;
  S: array[0..79] of Char;
begin
  ReadLn(S);
  P := StrNew(S);
  WriteLn(P);
  StrDispose(P);
end.
```

StrPas function — Strings

Function Converts a null-terminated string to a Pascal-style string.

Declaration

```
StrPas(Str: PChar)
```

Result type PChar

Remarks *StrPas* converts *Str* to a Pascal-style string.

See also *StrPCopy*

Example

```
uses Strings, WinCrt;
var
  A: array[0..79] of Char;
  S: string[79];
begin
  ReadLn(A);
  S := StrPas(A);
  WriteLn(S);
end.
```

StrPCopy function — Strings

Function Copies a Pascal-style string into a null-terminated string.

Declaration

```
StrPCopy(Dest: PChar; Source: String)
```

Result type PChar

Remarks *StrPCopy* copies the Pascal-style string *Source* into *Dest* and returns *Dest*. It is up to you to ensure that the buffer given by *Dest* has room for at least *Length(Source)* + 1 characters.

See also *StrCopy*

Example

```
uses Strings, WinCrt;
var
  A: array[0..79] of Char;
  S: string[79];
begin
  ReadLn(S);
  StrPCopy(A, S);
  WriteLn(A);
end.
```

StrPos function — Strings

Function Returns a pointer to the first occurrence of a string in another string.

Declaration

```
StrPos(Str1, Str2: PChar)
```

Result type PChar

Remarks *StrPos* returns a pointer to the first occurrence of *Str2* in *Str1*. If *Str2* does not occur in *Str1*, *StrPos* returns **nil**.

Example

```
uses Strings, WinCrt;
var
  P: PChar;
  S, SubStr: array[0..79] of Char;
begin
  ReadLn(S);
  ReadLn(SubStr);
  P := StrPos(S, SubStr);
  if P = nil then
    WriteLn('Substring not found');
  else
    WriteLn('Substring found at index ', P - S);
end.
```

StrRScan function — Strings

Function Returns a pointer to the last occurrence of a character in a string.

Declaration
```
StrRScan(Str: PChar; Chr: Char)
```

Result type PChar

Remarks *StrRScan* returns a pointer to the last occurrence of *Chr* in *Str*. If *Chr* does not occur in *Str*, *StrRScan* returns **nil**. The null terminator is considered to be part of the string.

See also *StrScan*

Example
```
{ Return pointer to name part of a full path name }

function NamePart(FileName: PChar): PChar;
var
  P: PChar;
begin
  P := StrRScan(FileName, '\');
  if P = nil then
  begin
    P := StrRScan(FileName, ':');
    if P = nil then P := FileName;
  end;
  NamePart := P;
end;
```

StrScan function — Strings

Function Returns a pointer to the first occurrence of a character in a string.

Declaration
```
StrScan(Str: PChar; Chr: Char)
```

Result type PChar

Remarks *StrScan* returns a pointer to the first occurrence of *Chr* in *Str*. If *Chr* does not occur in *Str*, *StrScan* returns **nil**. The null terminator is considered to be part of the string.

See also *StrRScan*

Example
```
{ Return true if file name has wildcards in it }

function HasWildcards(FileName: PChar): Boolean;
```

```
begin
  HasWildcards := (StrScan(FileName, '*') <> nil) or
    (StrScan(FileName, '?') <> nil);
end;
```

StrUpper function — Strings

Function Converts a string to uppercase.

Declaration `StrUpper(Str: PChar)`

Result type PChar

Remarks *StrUpper* converts *Str* to uppercase and returns *Str*.

See also *StrLower*

Example

```
uses Strings, WinCrt;
var
  S: array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn(StrUpper(S));
  WriteLn(StrLower(S));
end.
```

Succ function

Function Returns the successor of the argument.

Declaration `Succ(X)`

Result type Same type as parameter.

Remarks *X* is an ordinal-type expression. The result, of the same type as *X*, is the successor of *X*.

See also *Inc, Pred*

Swap function

Function Swaps the high- and low-order bytes of the argument.

Declaration `Swap(X)`

Result type Same type as parameter.

Remarks *X* is an expression of type Integer or Word.

See also *Hi, Lo*

Example

```
var
  X: Word;
begin
  X := Swap($1234);   { $3412 }
end.
```

TrackCursor procedure WinCrt

Function Scrolls the CRT window to ensure that the cursor is visible.

Declaration `TrackCursor;`

Trunc function

Function Truncates a real-type value to an integer-type value.

Declaration `Trunc(X: Real)`

Result type Longint

Remarks *X* is a real-type expression. *Trunc* returns a Longint value that is the value of *X* rounded toward zero.

Restrictions A run-time error occurs if the truncated value of *X* is not within the Longint range.

See also *Round, Int*

Truncate procedure

Function Truncates the file size at the current file position.

Declaration `Truncate(var F)`

Remarks *F* is a file variable of any type. All records past *F* are deleted and the current file position also becomes end-of-file (*Eof(F)* is True).

If I/O-checking is off, the *IOResult* function returns a nonzero value if an error occurs.

Restrictions *F* must be open. *Truncate* does not work on text files.

See also *Reset, Rewrite, Seek*

UnpackTime procedure — WinDos

Function Converts a 4-byte, packed date-and-time Longint returned by *GetFTime*, *FindFirst*, or *FindNext* into an unpacked *TDateTime* record.

Declaration `UnpackTime(Time: Longint; var DT: TDateTime)`

Remarks *TDateTime* is a record declared in the *WinDos* unit:

```
TDateTime = record
  Year, Month, Day, Hour, Min, Sec: Word
end;
```

The fields of the *Time* record are not range-checked.

See also *GetFTime, GetTime, PackTime, SetFTime, SetTime*

UpCase function

Function Converts a character to uppercase.

Declaration `UpCase(Ch: Char)`

Result type Char

Remarks *Ch* is an expression of type Char. The result of type Char is *Ch* converted to uppercase. Character values not in the range *a..z* are unaffected.

Val procedure

Function Converts the string value to its numeric representation.

Declaration

```
Val(S; var V; var Code: Integer)
```

Remarks *S* is a string-type expression or an expression of type *PChar* if the extended syntax is enabled. *V* is an integer-type or real-type variable. *Code* is a variable of type Integer. *S* must be a sequence of characters that form a signed whole number according to the syntax shown in the section "Numbers" in Chapter 1. *Val* converts *S* to its numeric representation and stores the result in *V*. If the string is somehow invalid, the index of the offending character is stored in *Code*; otherwise, *Code* is set to zero. For a null-terminated string, the error position returned in *Code* is one larger than the actual zero-based index of the character in error.

Val performs range-checking differently depending on the state of {**$R**} and the type of the parameter *V*.

With range-checking on, {**$R+**}, an out-of-range value always generates a run-time error. With range-checking off, {**$R-**}, the values for an out-of-range value vary depending upon the data type of *V*. If *V* is a Real or Longint type, the value of *V* is undefined and *Code* returns a nonzero value. For any other numeric type, *Code* returns a value of zero, and *V* will contain the results of an overflow calculation (assuming the string value is within the long integer range).

Therefore, you should pass *Val* a Longint variable and perform range-checking before making an assignment of the returned value:

```
{$R-}
Val('65536', LongIntVar, Code)
if (Code <> 0) or LongIntVar < 0) or (LongIntVar > 65535) then
  ...                                                    { Error }
else
  WordVar := LongIntVar;
```

In this example, *LongIntVar* would be set to 65,536, and *Code* would equal 0. Because 65,536 is out of range for a Word variable, an error would be reported.

Restrictions Trailing spaces must be deleted.

See also *Str*

Example

```
var I, Code: Integer;
```

```
begin
{ Get text from command line }
  Val(ParamStr(1), I, Code);
  { Error during conversion to integer? }
  if code <> 0 then
    Writeln('Error at position: ', Code)
  else
    Writeln('Value = ', I);
end.
```

WhereX procedure — WinCrt

Function Returns the X coordinate of the current cursor location.

Declaration `WhereX: Integer`

Remarks The returned value is 1-based, and corresponds to *Cursor.X* + 1.

WhereY procedure — WinCrt

Function Returns the Y coordinate of the currrent cursor location.

Declaration `WhereY: Integer`

Remarks The returned value is 1-based, and corresponds to *Cursor.Y* + 1.

Write procedure (text files)

Function Writes one or more values to a text file.

Declaration `Write( [ var F: Text; ] V1 [, V2,...,VN ] )`

Remarks *F*, if specified, is a text-file variable. If *F* is omitted, the standard file variable *Output* is assumed. Each *P* is a write parameter. Each write parameter includes an output expression whose value is to be written to the file. A write parameter can also contain the specifications of a field width and a number of decimal places. Each output expression must be of a type Char, Integer, Real, string, packed string, or Boolean.

A write parameter has the form

```
OutExpr [: MinWidth [: DecPlaces ] ]
```

where *OutExpr* is an output expression. *MinWidth* and *DecPlaces* are type integer expressions.

MinWidth specifies the minimum field width, which must be greater than 0. Exactly *MinWidth* characters are written (using leading blanks if necessary) except when *OutExpr* has a value that must be represented in more than *MinWidth* characters. In that case, enough characters are written to represent the value of *OutExpr*. Likewise, if *MinWidth* is omitted, then the necessary number of characters are written to represent the value of *OutExpr*.

DecPlaces specifies the number of decimal places in a fixed-point representation of a type Real value. It can be specified only if *OutExpr* is of type Real, and if *MinWidth* is also specified. When *MinWidth* is specified, it must be greater than or equal to 0.

Write with a type Char value: If *MinWidth* is omitted, the character value of *OutExpr* is written to the file. Otherwise, *MinWidth* – 1 blanks followed by the character value of *OutExpr* is written.

Write with a type integer value: If *MinWidth* is omitted, the decimal representation of *OutExpr* is written to the file with no preceding blanks. If *MinWidth* is specified and its value is larger than the length of the decimal string, enough blanks are written before the decimal string to make the field width *MinWidth*.

Write with a type real value: If *OutExpr* has a type real value, its decimal representation is written to the file. The format of the representation depends on the presence or absence of *DecPlaces*.

If *DecPlaces* is omitted (or if it is present, but has a negative value), a floating-point decimal string is written. If *MinWidth* is also omitted, a default *MinWidth* of 17 is assumed; otherwise, if *MinWidth* is less than 8, it is assumed to be 8. The format of the floating-point string is

```
[   | - ] <digit> . <decimals> E [ + | - ] <exponent>
```

The components of the output string are shown in Table 24.1:

Table 24.1
Components of the output string

[| -]	" " or "-", according to the sign of *OutExpr*
<digit>	Single digit, "0" only if *OutExpr* is 0
<decimals>	Digit string of *MinWidth-7* (but at most 10) digits
E	Uppercase [E] character
[+ | -]	According to sign of exponent
<exponent>	Two-digit decimal exponent

If *DecPlaces* is present, a fixed-point decimal string is written. If *DecPlaces* is larger than 11, it is assumed to be 11. The format of the fixed-point string follows:

```
[ <blanks> ] [ - ] <digits> [ . <decimals> ]
```

The components of the fixed-point string are shown in Table 24.2:

Table 24.2 Components of the fixed-point string

[<blanks>]	Blanks to satisfy *MinWidth*
[–]	If *OutExpr* is negative
<digits>	At least one digit, but no leading zeros
[. <decimals>]	Decimals if *DecPlaces* > 0

Write with a string-type value: If *MinWidth* is omitted, the string value of *OutExpr* is written to the file with no leading blanks. If *MinWidth* is specified, and its value is larger than the length of *OutExpr*, enough blanks are written before the decimal string to make the field width *MinWidth*.

Write with a packed string-type value: If *OutExpr* is of packed string type, the effect is the same as writing a string whose length is the number of elements in the packed string type.

Write with a Boolean value: If *OutExpr* is of type Boolean, the effect is the same as writing the strings True or False, depending on the value of *OutExpr*.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

Restrictions File must be open for output.

See also *Read, Readln, Writeln*

Write procedure (typed files)

Function Writes a variable into a file component.

Declaration

```
Write(F, V1 [, V2,...,VN ] )
```

Remarks *F* is a file variable, and each *V* is a variable of the same type as the component type of *F*. For each variable written, the current file position is advanced to the next component. If the current file position is at the end of the file—that is, if *Eof(F)* is True—the file is expanded.

With {**$I-**}, *IOResult* returns a 0 if the operation was successful; otherwise, it returns a nonzero error code.

See also *Writeln*

WriteBuf function — WinCrt

Function Writes a block of characters to the CRT window.

Declaration `WriteBuf(Buffer: PChar; Count: Word)`

Result type Word

Remarks *Buffer* points to the first character in the block, and *Count* contains the number of characters to write. If *AutoTracking* is *True*, the CRT window is scrolled if necessary to ensure that the cursor is visible after a block of characters is written.

WriteChar procedure — WinCrt

Function Writes a single character to the WinCrt window.

Declaration `WriteChar(Ch: Char)`

Remark Writes the character *Ch* to the WinCrt window at the current cursor position by calling *WriteBuf(@Ch, 1)*.

See also *WriteBuf*

Writeln procedure

Function Executes the *Write* procedure, then outputs an end-of-line marker to the file.

Declaration `Writeln( [ var F: Text; ] V1 [, V2,...,VN ] )`

Remarks *Writeln* procedure is an extension to the *Write* procedure, as it is defined for text files. After executing the Write, *Writeln* writes an end-of-line marker (carriage-return/line-feed) to the file. *Writeln(F)* with no parameters writes an end-of-line marker to the file. (*Writeln* with no parameter list altogether corresponds to *Writeln(Output)*.)

Restrictions File must be open for output.

See also *Write*

PART 6

Appendixes

A P P E N D I X

A

Error messages

Compiler error messages

The following lists the possible error messages you can get from the compiler during program development. Whenever possible, the compiler will display additional diagnostic information in the form of an identifier or a file name. For example,

```
Error 15: File not found (WINDOW.TPU).
```

When an error is detected, Turbo Pascal (in the IDE) automatically loads the source file and places the cursor at the error. The command-line compiler displays the error message and number and the source line, and uses a caret (^) to indicate where the error occurred. Note, however, that some errors are not detected until a little later in the source text. For example, a type mismatch in an assignment statement cannot be detected until the entire expression after the := has been evaluated. In such cases, look for the error to the left of or above the cursor.

1 Out of memory.

This error occurs when the compiler has run out of memory. There are a number of possible solutions to this problem:

- Try to increase the amount of available memory in Windows. For example, you could terminate other applications that are currently running.

- If Options | Linker | Link Buffer is set to Memory, toggle it to Disk. Use a **/L** option to place the link buffer on disk when using the command-line compiler.

If these suggestions don't help, your program or unit may simply be too large to compile in the amount of memory available, and you may have to break it into two or more smaller units.

2 Identifier expected.

An identifier was expected at this point. You may be trying to redeclare a reserved word.

3 Unknown identifier.

This identifier has not been declared, or may not be visible within the current scope.

4 Duplicate identifier.

The identifier has already been used within the current block.

5 Syntax error.

An illegal character was found in the source text. You may have forgotten the quotes around a string constant.

6 Error in real constant.

The syntax of real-type constants is defined in Chapter 1, "Tokens and constants."

7 Error in integer constant.

The syntax of integer-type constants is defined in Chapter 1, "Tokens and constants." Note that whole real numbers outside the maximum integer range must be followed by a decimal point and a zero; for example, 12,345,678,912.0.

8 String constant exceeds line.

You have most likely forgotten the ending quote in a string constant.

9 Too many nested files.

The compiler allows no more than 15 nested source files. Most likely you have more than four nested Include files.

10 Unexpected end of file.

You might have gotten this error message because of one of the following:

- Your source file ends before the final **end** of the main statement part. Most likely, your **begin**s and **end**s are unbalanced.
- An Include file ends in the middle of a statement part. Every statement part must be entirely contained in one file.
- You didn't close a comment.

11 Line too long.

The maximum line length is 126 characters.

12 Type identifier expected.

The identifier does not denote a type as it should.

13 Too many open files.

If this error occurs, your CONFIG.SYS file does not include a FILES=*xx* entry or the entry specifies too few files. Increase the number to some suitable value, for instance, 20.

14 Invalid file name.

The file name is invalid or specifies a nonexistent path.

15 File not found.

The file could not be found in the current directory or in any of the search directories that apply to this type of file.

16 Disk full.

Delete some files or use a new disk.

17 Invalid compiler directive.

The compiler directive letter is unknown, one of the compiler directive parameters is invalid, or you are using a global compiler directive when compilation of the body of the program has begun.

18 Too many files.

There are too many files involved in the compilation of the program or unit. Try not to use that many files, for instance, by merging Include files or making the file names shorter.

19 Undefined type in pointer definition.

The type was referenced in a pointer-type declaration previously, but it was never declared.

20 Variable identifier expected.

The identifier does not denote a variable as it should.

21 Error in type.

This symbol cannot start a type definition.

22 Structure too large.

The maximum allowable size of a structured type is 65,520 bytes.

23 Set base type out of range.

The base type of a set must be a subrange with bounds in the range 0..255 or an enumerated type with no more than 256 possible values.

24 File components may not be files or objects.

file of file and **file of object** constructs are not allowed; nor is any structured type that includes an object type or file type.

25 Invalid string length.

The declared maximum length of a string must be in the range 1..255.

26 Type mismatch.

This is due to one of the following:

- incompatible types of the variable and the expression in an assignment statement
- incompatible types of the actual and formal parameter in a call to a procedure or function
- an expression type that is incompatible with index type in array indexing
- incompatible types of operands in an expression

27 Invalid subrange base type.

All ordinal types are valid base types.

28 Lower bound greater than upper bound.

The declaration of a subrange type specifies a lower bound greater than the upper bound.

29 Ordinal type expected.

Real types, string types, structured types, and pointer types are not allowed here.

30 Integer constant expected.

31 Constant expected.

32 Integer or real constant expected.

33 Pointer type identifier expected.

The identifier does not denote a pointer type as it should.

34 Invalid function result type.

Valid function result types are all simple types, string types, and pointer types.

35 Label identifier expected.

The identifier does not denote a label as it should.

36 BEGIN expected.

A **begin** is expected here, or there is an error in the block structure of the unit or program.

37 END expected.

An **end** is expected here, or there is an error in the block structure of the unit or program.

38 Integer expression expected.

The preceding expression must be of an integer type.

39 Ordinal expression expected.

The preceding expression must be of an ordinal type.

40 Boolean expression expected.

The preceding expression must be of type boolean.

41 Operand types do not match operator.

The operator cannot be applied to operands of this type, for example, '*A*' **div** '2'.

42 Error in expression.

This symbol cannot participate in an expression in the way it does. You may have forgotten to write an operator between two operands.

43 Illegal assignment.

- Files and untyped variables cannot be assigned values.
- A function identifier can only be assigned values within the statement part of the function.

44 Field identifier expected.

The identifier does not denote a field in the preceding record variable.

45 Object file too large.

Turbo Pascal cannot link in .OBJ files larger than 64K.

46 Undefined external.

The **external** procedure or function did not have a matching **PUBLIC** definition in an object file. Make sure you have specified all object files in {**$L** *filename*} directives, and check the spelling of the procedure or function identifier in the .ASM file.

47 Invalid object file record.

The .OBJ file contains an invalid object record; make sure the file is in fact an .OBJ file.

48 Code segment too large.

The maximum size of the code of a program or unit is 65,520 bytes. If you are compiling a program, move some procedures or functions into a unit. If you are compiling a unit, break it into two or more units.

49 Data segment too large.

The maximum size of a program's data segment is 65,520 bytes, including data declared by the used units. If you need more global data than this, declare the larger structures as pointers, and allocate them dynamically using the *New* procedure.

50 DO expected.

The reserved word **do** does not appear where it should.

51 Invalid PUBLIC definition.

- Two or more **PUBLIC** directives in assembly language define the same identifier.
- The .OBJ file defines **PUBLIC** symbols that do not reside in the **CODE** segment.

52 Invalid EXTRN definition.

- The identifier was referred to through an **EXTRN** directive in assembly language, but it is not declared in the Pascal program or unit, nor in the interface part of any of the used units.
- The identifier denotes an **absolute** variable.
- The identifier denotes an **inline** procedure or function.

53 Too many EXTRN definitions.

Turbo Pascal cannot handle .OBJ files with more than 256 **EXTRN** definitions.

54 OF expected.

The reserved word **of** does not appear where it should.

55 INTERFACE expected.

The reserved word **interface** does not appear where it should.

56 Invalid relocatable reference.

- The .OBJ file contains data and relocatable references in segments other than **CODE**. For example, you are attempting to declare initialized variables in the **DATA** segment.
- The .OBJ file contains byte-sized references to relocatable symbols. This error occurs if you use the **HIGH** and **LOW** operators with relocatable symbols or if you refer to relocatable symbols in **DB** directives.
- An operand refers to a relocatable symbol that was not defined in the **CODE** segment or in the **DATA** segment.
- An operand refers to an **EXTRN** procedure or function with an offset, for example, **CALL** *SortProc+8*.

57 THEN expected.

The reserved word **then** does not appear where it should.

58 TO or DOWNTO expected.

The reserved word **to** or **downto** does not appear where it should.

59 Undefined forward.

- The procedure or function was declared in the **interface** part of a unit, but its definition never occurred in the **implementation** part.
- The procedure or function was declared with **forward**, but its definition was never found.

61 Invalid typecast.

- The sizes of the variable reference and the destination type differ in a variable typecast.
- You are attempting to typecast an expression where only a variable reference is allowed.

62 Division by zero.

The preceding operand attempts to divide by zero.

63 Invalid file type.

The file type is not supported by the file-handling procedure; for example, *Readln* with a typed file or *Seek* with a text file.

64 Cannot Read or Write variables of this type.

- *Read* and *Readln* can input variables of Char, integer, real, and string types.
- *Write* and *Writeln* can output variables of Char, integer, real, string, and Boolean types.

65 Pointer variable expected.

The preceding variable must be of a pointer type.

66 String variable expected.

The preceding variable must be of a string type.

67 String expression expected.

The preceding expression must be of a string type.

68 Circular unit reference.

Two units are not allowed to use each other:

```
unit U1;          unit U2;
uses U2;          uses U1;
...               ...
```

In this example, doing a Make on either unit generates error 68.

69 Unit name mismatch.

The name of the unit found in the .TPU file does not match the name specified in the **uses** clause.

70 Unit version mismatch.

One or more of the units used by this unit have been changed since the unit was compiled. Use **C**ompile | **M**ake or **C**ompile | **B**uild in the IDE and **/M** or **/B** options in the command-line compiler to automatically compile units that need recompilation.

72 Unit file format error.

The .TPU file is somehow invalid; make sure it is in fact a .TPU file.

73 IMPLEMENTATION expected.

The reserved word **implementation** does not appear where it should.

74 Constant and case types do not match.

The type of the **case** constant is incompatible with the **case** statement's selector expression.

75 Record variable expected.

The preceding variable must be of a record type.

76 Constant out of range.

You are trying to

- index an array with an out-of-range constant
- assign an out-of-range constant to a variable
- pass an out-of-range constant as a parameter to a procedure or function

77 File variable expected.

The preceding variable must be of a file type.

78 Pointer expression expected.

The preceding expression must be of a pointer type.

79 Integer or real expression expected.

The preceding expression must be of an integer or a real type.

80 Label not within current block.

A **goto** statement cannot reference a label outside the current block.

81 Label already defined.

The label already marks a statement.

82 Undefined label in preceding statement part.

The label was declared and referenced in the preceding statement part, but it was never defined.

83 Invalid @ argument.

Valid arguments are variable references and procedure or function identifiers.

84 UNIT expected.

The reserved word **unit** does not appear where it should.

85 ";" expected.

A semicolon does not appear where it should.

86 ":" expected.

A colon does not appear where it should.

87 "," expected.

A comma does not appear where it should.

88 "(" expected.

An opening parenthesis does not appear where it should.

89 ")" expected.

A closing parenthesis does not appear where it should.

90 "=" expected.

An equal sign does not appear where it should.

91 ":=" expected.

An assignment operator does not appear where it should.

92 "[" or "(." expected.

A left bracket does not appear where it should.

93 "]" or ".)" expected.

A right bracket does not appear where it should.

94 "." expected.

A period does not appear where it should.

95 ".." expected.

A subrange does not appear where it should.

96 Too many variables.

- The total size of the global variables declared within a program or unit cannot exceed 64K.
- The total size of the local variables declared within a procedure or function cannot exceed 64K.

97 Invalid FOR control variable.

The **for** statement control variable must be a simple variable defined in the declaration part of the current subprogram.

98 Integer variable expected.

The preceding variable must be of an integer type.

99 File and procedure types are not allowed here.

A typed constant cannot be of a file or procedural type.

100 String length mismatch.

The length of the string constant does not match the number of components in the character array.

101 Invalid ordering of fields.

The fields of a record-type constant must be written in the order of declaration.

102 String constant expected.

A string constant does not appear where it should.

103 Integer or real variable expected.

The preceding variable must be of an integer or real type.

104 Ordinal variable expected.

The preceding variable must be of an ordinal type.

105 INLINE error.

The **<** operator is not allowed in conjunction with relocatable references to variables—such references are always word-sized.

106 Character expression expected.

The preceding expression must be of a Char type.

112 CASE constant out of range.

For integer type **case** statements, the constants must be within the range –32,768..32,767.

113 Error in statement.

This symbol cannot start a statement.

114 Cannot call an interrupt procedure.

You cannot directly call an interrupt procedure.

116 Must be in 8087 mode to compile this.

This construct can only be compiled in the {**$N+**} state. Operations on the 8087 real types (Single, Double, Extended, and Comp) are not allowed in the {**$N-**} state.

117 Target address not found.

The Search | Find Error command in the IDE or the **/F** option in the command-line version could not locate a statement that corresponds to the specified address.

118 Include files are not allowed here.

Every statement part must be entirely contained in one file.

120 NIL expected.

Typed constants of pointer types may only be initialized to the value **nil**.

121 Invalid qualifier.

You are trying to do one of the following:

- index a variable that is not an array
- specify fields in a variable that is not a record
- dereference a variable that is not a pointer

122 Invalid variable reference.

The preceding construct follows the syntax of a variable reference, but it does not denote a memory location. Most likely, you are calling a pointer function, but forgetting to dereference the result.

123 Too many symbols.

The program or unit declares more than 64K of symbols. If you are compiling with {**$D+**}, try turning it off—note, however, that this will prevent you from finding run-time errors in that module. Otherwise, you could try moving some declarations into a separate unit.

124 Statement part too large.

Turbo Pascal limits the size of a statement part to about 24K. If you encounter this error, move sections of the statement part into one or more procedures. In any case, with a statement part of that size, it's worth the effort to clarify the structure of your program.

126 Files must be var parameters.

You are attempting to declare a file-type value parameter. File-type parameters must be **var** parameters.

127 Too many conditional symbols.

There is not enough room to define further conditional symbols. Try to eliminate some symbols, or shorten some of the symbolic names.

128 Misplaced conditional directive.

The compiler encountered an {**$ELSE**} or {**$ENDIF**} directive without a matching {**$IFDEF**}, {**$IFNDEF**}, or {**$IFOPT**} directive.

129 ENDIF directive missing.

The source file ended within a conditional compilation construct. There must be an equal number of {**$IF*xxx***}s and {**$ENDIF**}s in a source file.

130 Error in initial conditional defines.

The initial conditional symbols specified in **O**ptions | **C**ompiler | **C**onditional Defines (in the IDE) or in a **/D** directive (with the command-line compiler) are invalid. Turbo Pascal expects zero or more identifiers separated by blanks, commas, or semicolons.

131 Header does not match previous definition.

The procedure or function header specified in the **interface** part or **forward** declaration does not match this header.

132 Critical disk error.

A critical error occurred during compilation (for example, drive not ready error).

133 Cannot evaluate this expression.

You are attempting to use a non-supported feature in a constant expression. For example, you're attempting to use the *Sin* function in a **const** declaration. For a description of the allowed syntax of constant expressions, refer to Chapter 1, "Tokens and constants."

136 Invalid indirect reference.

The statement attempts to make an invalid indirect reference. For example, you are using an **absolute** variable whose base variable is not known in the current module, or you are using an **inline** routine that references a variable not known in the current module.

137 Structured variables are not allowed here.

You are attempting to perform a non-supported operation on a structured variable. For example, you are trying to multiply two records.

140 Invalid floating-point operation.

An operation on two real type values produced an overflow or a division by zero.

142 Procedure or function variable expected.

In this context, the address operator (**@**) can only be used with a procedure or function variable.

143 Invalid procedure or function reference.

- You are attempting to call a procedure in an expression.

- If you are going to assign a procedure or function to a procedural variable, it must be compiled in the {**$F+**} state and cannot be declared with **inline** or **interrupt**.

146 File access denied

The file could not be opened or created. Most likely, the compiler is trying to write to a read-only file.

147 Object type expected.

The identifier does not denote an object type.

148 Local object types are not allowed.

Object types can be defined only in the outermost scope of a program or unit. Object-type definitions within procedures and functions are not allowed.

149 VIRTUAL expected.

The reserved word **virtual** is missing.

150 Method identifier expected.

The identifier does not denote a method.

151 Virtual constructors are not allowed.

A constructor method must be static.

152 Constructor identifier expected.

The identifier does not denote a constructor.

153 Destructor identifier expected.

The identifier does not denote a destructor.

154 Fail only allowed within constructors.

The *Fail* standard procedure can be used only within constructors.

155 Invalid combination of opcode and operands.

The assembler opcode does not accept this combination of operands. Possible causes are:

- There are too many or too few operands for this assembler opcode; for example, **INC AX,BX** or **MOV AX**.
- The number of operands is correct, but their types or order do not match the opcode; for example, **DEC 1**, **MOV AX,CL** or **MOV 1,AX**.

156 Memory reference expected.

The assembler operand is not a memory reference, which is required here. Most likely you have forgotten to put square brackets around an index register operand, for example, **MOV AX,BX+SI** instead of **MOV AX,[BX+SI]**.

157 Cannot add or subtract relocatable symbols.

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols. Assuming that *Var* is a variable and *Const* is a constant, then the instructions `MOV AX,Const+Const` and `MOV AX,Var+Const` are valid, but `MOV AX,Var+Var` is not.

158 Invalid register combination.

Valid index register combinations are **[BX]**, **[BP]**, **[SI]**, **[DI]**, **[BX+SI]**, **[BX+DI]**, **[BP+SI]**, and **[BP+DI]**. Other index register combinations (such as **[AX]**, **[BP+BX]**, and **[SI+DX]**) are not allowed.

Local variables (variables declared in procedures and functions) are always allocated on the stack and accessed via the BP register. The assembler automatically adds **[BP]** in references to such variables, so even though a construct like **Local[BX]** (where **Local** is a local variable) appears valid, it is not since the final operand would become **Local[BP+BX]**.

159 286/287 instructions are not enabled.

Use a {**$G+**} compiler directive to enable 286/287 opcodes, but be aware that the resulting code cannot be run on 8086 and 8088-based machines.

160 Invalid symbol reference.

This symbol cannot be accessed in an assembler operand. Possible causes follow:

- You are attempting to access a standard procedure, a standard function, or the *Mem*, *MemW*, *MemL*, *Port*, or *PortW* special arrays in an assembler operand.
- You are attempting to access a string, floating-point, or set constant in an assembler operand.
- You are attempting to access an **inline** procedure or function in an assembler operand.
- You are attempting to access the *@Result* special symbol outside a function.
- You are attempting to generate a short **JMP** instruction that jumps to something other than a label.

161 Code generation error.

The preceding statement part contains a **LOOPNE**, **LOOPE**, **LOOP**, or **JCXZ** instruction that cannot reach its target label.

162 ASM expected.

163 Duplicate dynamic method index

This dynamic method index has already been used by another method. Perhaps you're trying to override a dynamic method but have misspelled its name, thus instead introducing a new method.

164 Duplicate resource identifier

This resource file contains a resource with a name or ID that has already been used by another resource.

165 Duplicate or invalid export index

The ordinal number specified in the **index** clause is not within the range 1..32767, or it has already been used by another exported routine.

166 Procedure or function identifier expected

The **exports** clause only allows procedures and functions to be exported.

167 Cannot export this symbol

A procedure or function cannot be exported unless it was declared with the **export** procedure directive.

168 Duplicate export name

The name specified in the **name** clause has already been used by another exported routine.

Run-time errors

Certain errors at run time cause the program to display an error message and terminate:

```
Run-time error nnn at xxxx:yyyy
```

where *nnn* is the run-time error number, and *xxxx:yyyy* is the run-time error address (segment and offset).

The run-time errors are divided into three categories: DOS errors 1 through 99; I/O errors, 100 through 199; and fatal errors, 200 through 255.

DOS errors

1 Invalid function number.

You made a call to a nonexistent DOS function.

2 File not found.

Reported by *Reset*, *Append*, *Rename*, or *Erase* if the name assigned to the file variable does not specify an existing file.

3 Path not found.

- Reported by *Reset*, *Rewrite*, *Append*, *Rename*, or *Erase* if the name assigned to the file variable is invalid or specifies a nonexistent subdirectory.

- Reported by *ChDir*, *MkDir*, or *RmDir* if the path is invalid or specifies a nonexistent subdirectory.

4 Too many open files.

Reported by *Reset*, *Rewrite*, or *Append* if the program has too many open files. DOS never allows more than 15 open files per process. If you get this error with less than 15 open files, it may indicate that the CONFIG.SYS file does not include a FILES=*xx* entry or that the entry specifies too few files. Increase the number to some suitable value, for instance, 20.

5 File access denied.

- Reported by *Reset* or *Append* if *FileMode* allows writing and the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *Rewrite* if the directory is full or if the name assigned to the file variable specifies a directory or an existing read-only file.
- Reported by *Rename* if the name assigned to the file variable specifies a directory or if the new name specifies an existing file.
- Reported by *Erase* if the name assigned to the file variable specifies a directory or a read-only file.
- Reported by *MkDir* if a file with the same name exists in the parent directory, if there is no room in the parent directory, or if the path specifies a device.
- Reported by *RmDir* if the directory isn't empty, if the path doesn't specify a directory, or if the path specifies the root directory.
- Reported by *Read* or *BlockRead* on a typed or untyped file if the file is not open for reading.
- Reported by *Write* or *BlockWrite* on a typed or untyped file if the file is not open for writing.

6 Invalid file handle.

This error is reported if an invalid file handle is passed to a DOS system call. It should never occur; if it does, it is an indication that the file variable is somehow trashed.

12 Invalid file access code.

Reported by *Reset* or *Append* on a typed or untyped file if the value of *FileMode* is invalid.

15 Invalid drive number.

Reported by *GetDir* or *ChDir* if the drive number is invalid.

16 Cannot remove current directory.

Reported by *RmDir* if the path specifies the current directory.

17 Cannot rename across drives.

Reported by *Rename* if both names are not on the same drive.

I/O errors

These errors cause termination if the particular statement was compiled in the {**$I+**} state. In the {**$I-**} state, the program continues to execute, and the error is reported by the *IOResult* function.

100 Disk read error.

Reported by *Read* on a typed file if you attempt to read past the end of the file.

101 Disk write error.

Reported by *Close, Write, Writeln, Flush,* or *Page* if the disk becomes full.

102 File not assigned.

Reported by *Reset, Rewrite, Append, Rename,* and *Erase* if the file variable has not been assigned a name through a call to *Assign.*

103 File not open.

Reported by *Close, Read, Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead,* or *BlockWrite* if the file is not open.

104 File not open for input.

Reported by *Read, Readln, Eof, Eoln, SeekEof,* or *SeekEoln* on a text file if the file is not open for input.

105 File not open for output.

Reported by *Write* and *Writeln* on a text file if the file is not open for output.

106 Invalid numeric format.

Reported by *Read* or *Readln* if a numeric value read from a text file does not conform to the proper numeric format.

Fatal errors

These errors always immediately terminate the program.

200 Division by zero.

The program attempted to divide a number by zero during a **/**, **mod**, or **div** operation.

201 Range check error.

This error is reported by statements compiled in the {**$R+**} state when one of the following situations arises:

- The index expression of an array qualifier was out of range.
- You attempted to assign an out-of-range value to a variable.
- You attempted to assign an out-of-range value as a parameter to a procedure or function.

202 Stack overflow error.

This error is reported on entry to a procedure or function compiled in the {**$S+**} state when there is not enough stack space to allocate the subprogram's local variables. Increase the size of the stack by using the **$M** compiler directive.

This error may also be caused by infinite recursion, or by an assembly language procedure that does not maintain the stack properly.

203 Heap overflow error.

This error is reported by *New* or *GetMem* when there is not enough free space in the heap to allocate a block of the requested size.

For a complete discussion of the heap manager, see Chapter 16, "Memory issues."

204 Invalid pointer operation.

This error is reported by *Dispose* or *FreeMem* if the pointer is **nil** or points to a location outside the heap.

205 Floating point overflow.

A floating-point operation produced a number too large for Turbo Pascal or the numeric coprocessor (if any) to handle.

206 Floating point underflow

A floating-point operation produced an underflow. This error is only reported if you are using the 8087 numeric coprocessor with a control word that unmasks underflow exceptions. By default, an underflow causes a result of zero to be returned.

207 Invalid floating point operation

- The real value passed to *Trunc* or *Round* could not be converted to an integer within the Longint range (-2,147,483,648 to 2,147,483,647).
- The argument passed to the *Sqrt* function was negative.
- The argument passed to the *Ln* function was zero or negative.
- An 8087 stack overflow occurred. For further details on correctly programming the 8087, refer to Chapter 15, "Using the 80x87."

210 Object not initialized

With range-checking on, you made a call to an object's virtual method, before the object had been initialized via a constructor call.

211 Call to abstract method.

This error is generated by the *Abstract* procedure in the *WObjects* unit; it indicates that your program tried to execute an abstract virtual method. When an object type contains one or more abstract methods it is called an *abstract object type*. It is an error to instantiate objects of an abstract type—abstract object types exist

only so that you can inherit from them and override the abstract methods.

For example, the *Compare* method of the *TSortedCollection* type in the *WObjects* unit is abstract, indicating that to implement a sorted collection you must create an object type that inherits from *TSortedCollection* and overrides the *Compare* method.

212 Stream registration error.

This error is generated by the *RegisterType* procedure in the *WObjects* unit indicating that one of the following errors have occurred:

- The stream registration record does not reside in the data segment.
- The *ObjType* field of the stream registration record is zero.
- The type has already been registered.
- Another type with the same *ObjType* value already exists.

213 Collection index out of range.

The index passed to a method of a *TCollection* is out of range.

214 Collection overflow error.

The error is reported by a *TCollection* if an attempt is made to add an element when the collection cannot be expanded.

A P P E N D I X

B

Reference materials

This appendix is devoted to certain reference materials: an ASCII table and keyboard scan codes.

ASCII codes

The American Standard Code for Information Interchange (ASCII) is a code that translates alphabetic and numeric characters and symbols and control instructions into 7-bit binary code. Table B.1 shows both printable characters and control characters.

Table B.1
ASCII table

The caret in ^@ means to press the Ctrl key and type @.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	^@ NUL	32	20		64	40	@	96	60	'
1	1	☺ SOH	33	21	!	65	41	A	97	61	a
2	2	● STX	34	22	"	66	42	B	98	62	b
3	3	♥ ETX	35	23	#	67	43	C	99	63	c
4	4	♦ EOT	36	24	$	68	44	D	100	64	d
5	5	♣ ENQ	37	25	%	69	45	E	101	65	e
6	6	♠ ACK	38	26	&	70	46	F	102	66	f
7	7	• BEL	39	27	'	71	47	G	103	67	g
8	8	◘ BS	40	28	(	72	48	H	104	68	h
9	9	○ TAB	41	29	)	73	49	I	105	69	i
10	A	◙ LF	42	2A	*	74	4A	J	106	6A	j
11	B	♂ VT	43	2B	+	75	4B	K	107	6B	k
12	C	♀ FF	44	2C	,	76	4C	L	108	6C	l
13	D	♪ CR	45	2D	-	77	4D	M	109	6D	m
14	E	♫ SO	46	2E	.	78	4E	N	110	6E	n
15	F	☼ SI	47	2F	/	79	4F	O	111	6F	o
16	10	► DLE	48	30	0	80	50	P	112	70	p
17	11	◄ DC1	49	31	1	81	51	Q	113	71	q
18	12	↕ DC2	50	32	2	82	52	R	114	72	r
19	13	‼ DC3	51	33	3	83	53	S	115	73	s
20	14	¶ DC4	52	34	4	84	54	T	116	74	t
21	15	§ NAK	53	35	5	85	55	U	117	75	u
22	16	▬ SYN	54	36	6	86	56	V	118	76	v
23	17	↨ ETB	55	37	7	87	57	W	119	77	w
24	18	↑ CAN	56	38	8	88	58	X	120	78	x
25	19	↓ EM	57	39	9	89	59	Y	121	79	y
26	1A	→ SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	← ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	└ FS	60	3C	<	92	5C	\	124	7C	|
29	1D	↔ GS	61	3D	=	93	5D	]	125	7D	}
30	1E	▲ RS	62	3E	>	94	5E	^	126	7E	~
31	1F	▼ US	63	3F	?	95	5F	_	127	7F	⌂

Table B.1: ASCII table (continued)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	⣿	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	n
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	2
158	9E	Pt	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Keyboard scan codes

Keyboard scan codes are the codes returned from the keys on the IBM PC keyboard, as they are seen by Turbo Pascal. These keys are useful when you're working at the assembly language level. Note that the keyboard scan codes displayed in Table B.2 are in hexadecimal values.

Table B.2
Keyboard scan codes

Key	Scan Code in Hex	Key	Scan Code in Hex
Esc	01	←/→	0F
! 1	02	Q	10
@ 2	03	W	11
# 3	04	E	12
$ 4	05	R	13
% 5	06	T	14
^ 6	07	Y	15
& 7	08	U	16
* 8	09	I	17
(9	0A	O	18
) 0	0B	P	19
_ -	0C	{ [	1A
+=	0D	}]	1B
Backspace	0E	Return	1C
Ctrl	1D	| \	2B
A	1E	Z	2C
S	1F	X	2D
D	20	C	2E
F	21	V	2F
G	22	B	30
H	23	N	31
J	24	M	32
K	25	< ,	33
L	26	> .	34
: ;	27	? /	35
" '	28	→Shift	36
~ `	29	PrtSc*	37
←Shift	2A	Alt	38
Spacebar	39	7 Home	47
Caps Lock	3A	8 ↑	48
F1	3B	9 PgUp	49
F2	3C	Minus sign	4A
F3	3D	4 ←	4B
F4	3E	5	4C
F5	3F	6 →	4D
F6	40	+	4E
F7	41	1 End	4F
F8	42	2 ↓	50
F9	43	3 PgDn	51
F10	44	0 Ins	52
F11	D9	Del	53
F12	DA	Num Lock	45
Scroll Lock	46		

I N D E X

$ *See* compiler, directives
8087/80287/387 coprocessor *See* numeric coprocessor
80286 code generation compiler switch *249*
286 Code option *249*
@ (address-of) operator *See* address-of (@) operator
^ (pointer) symbol *41, 53*
# (pound) character *11*
80x87 code option *256*
80x87 emulation *187*

A

$A compiler directive *241, 247*
Abs function *143, 239, 306*
absolute clause *49*
activation, qualified *85*
actual parameters *77, 85*
Addr function *144, 306*
address functions *144*
address-of (@) operator *41, 53, 75, 81*
  double *81*
  versus Addr *306*
  with method designators *77*
address of object *306*
Align Data command *247*
aligning data *247*
ancestors *33*
and operator *71*
Append procedure *145, 148, 307*
ArcTan function *143, 308*
argument size *363*
arithmetic
  functions *143*
  operators *69*
array-type constants *59*
arrays *29, 51, 59*
  types *29, 200*
  variables *51*
  zero-based character *60, 162, 163*
ASCII codes *413*
.ASM files *187*
assembly language *255, 291*
  80x87 emulation and *187*
  examples *293*
  inline
    directives *300*
    statements *298*
  interfacing program routines with *292*
Assign procedure *145, 146, 235, 308*
AssignCrt procedure *176, 180, 235, 309*
assigning text to CRT window *309*
assignment
  compatibility *39, 84*
  statements *84*
automatic
  call model selection, overriding *228*
  word alignment *241*
automatic data segment *192*
AutoTracking variable *172*
AX register *227, 300*

B

$B compiler directive *247*
binding
  late *36*
bitwise operators *70*
BlockRead procedure *149, 309*
blocks, program *15*
BlockWrite procedure *149, 311*
Boolean
  evaluation, compiler switch *247*
  operators *70*
  types *24*
    Boolean *24, 197*
    LongBool *24, 197*

WordBool *24, 197*
values *24*
Boolean evaluation
short circuit *247*
Boolean Evaluation command *247*
BP register *231, 233, 299*
brackets, in expressions *78*
buffer
assign to text file *360*
buffers, flushing *328*
BX register *227, 233*
Byte data type *23*

C

$C code segment attribute *254*
calling conventions *225*
constructors and destructors *214*
methods *85, 214*
case statements *89*
Char data type *25, 197*
character
convert to uppercase *378*
character pointer indexing *164*
character pointer operations *166*
characters
strings *11*
ChDir procedure *146, 312*
CheckEOF variable *173*
Chr function *142, 239, 312*
circular unit references *122*
clearing the screen *313*
clearing to end of line *313*
Close function *237*
Close procedure *146, 235, 313*
ClrEol procedure *176, 179, 313*
ClrScr procedure *176, 179, 313*
CmdLine variable *151*
CmdShow variable *151*
CODE *291*
Code generation
Windows *252*
code segment
maximum size *191*
Code segment attribute *254*
code segment attributes *191*
changing *192*
command-line parameter *330*
command-line parameters *144, 346*
comments
inline assembler *266, 267*
program *13*
common type *24*
Comp floating-point type *183, 199*
comparing values of real types *185*
compatibility
assignment *39, 84*
parameter type *110*
compilation, conditional *257*
compiler
directives *13, 245-261*
$A *241, 247*
$B *247*
Boolean evaluation *247*
$C *254*
conditional *245, 257-261*
$D *248, 255*
$DEFINE *258, 260*
$ELSE *261*
$ENDIF *261*
$F *99, 228, 249*
$G *249*
$I *146, 250, 255, 339*
$IFDEF *260*
$IFNDEF *261*
$IFOPT *261*
$L *250, 255, 291*
local symbol *250*
$M *48, 256, 341*
$N *27, 182, 187, 256*
parameter *245*
$R *251, 257*
$S *48, 251*
switch *245, 246-252*
$UNDEF *258, 260*
$V *252*
$W *252*
$X *253*
error messages *387*
compiler directives
$M *342*
Complete Boolean Eval option *247*
compound statements *87*
Concat function *144, 314*
concatenating strings *314*

concatenation *72*
conditional
  compilation *257*
  statements *88*
  symbols *258*
CONST *291*
constant expressions *12*
  restrictions *12*
  type definition *26*
constants *153*
  array-type *59*
  declaration part *16*
  declarations *12*
  file attribute *154*
  file name component *155*
  folding *239*
  inline assembler *275, 281*
  merging *240*
  pointer-type *62*
  procedural-type *63*
  record-type *61*
  set-type *62*
  simple *12*
  simple-type *58*
  string-type *59*
  structured-type *59*
  typed *57*
    object type *62*
constructors
  calling conventions *214*
  declaring *105*
  defined *106*
  error recovery *220*
  implementation *105*
  inherited *37*
  virtual methods and *38, 106*
  VMTP and *205, 208*
control-break checking *330*
  setting *358*
control characters *11, 413*
  CRT window in *173*
Copy function *144, 314*
copying substrings *314*
Cos function *143, 315*
CPU symbols *259, 260*
CreateDir procedure *159, 315*
CRT window *172*
  closing the *172*
  control characters in *173*
  create the *337*
  line input *173*
  read a line in *349*
  scrolling the *172, 356*
  write block to *383*
CS register *233, 315*
  value in *315*
CSEG *291*
CSeg function *144, 315*
cursor
  location
    X coordinate *380*
    Y coordinate *380*
  moving the *335*
  tracking the *377*
CursorTo procedure *176, 179, 316*
CX register *233*

D

$D compiler directive *248*
$D description *255*
DATA *291*
data
  alignment *247*
  encryption *152*
  ports *237*
  segment *48*
  types *See* types
date and time procedures *158*
  GetDate *331*
  GetFTime *333*
  GetTime *334*
  SetDate *359*
  SetFTime *360*
  SetTime *362*
dead code removal *242*
Debug Information
  command *248*
  option *248*
debugging
  information switch *248*
  range-checking switch *251*
  run-time error messages *406*
  stack overflow switch *251*
Dec procedure *143, 316*

decimal notation *9*
declaration
  constructors *105*
  destructors *105*
  methods *36, 105*
  object types *34*
  part
    block *15*
$DEFINE compiler directive *258, 260*
Delete procedure *143, 316*
DEMANDLOAD code segment attribute *192*
descendants *33*
Description directive *255*
designators
  field *53*
  method *39, 53*
    @ (address operator) with *77*
destroy CRT window *318*
destructors *106*
  calling conventions *214*
  declaring *105*
  defined *106*
  implementation *105*
devices *149*
  drivers *235*
  handlers *233, 234*
DI register *233*
direct memory *202*
directives *7, See* compiler, directives
  external *101*
  far *99*
  forward *100*
  inline *102*
  inline assembler *265, 287, 288*
  interrupt *100*
  near *99*
  private *7*
directories *331*
  changing *312, 358*
  creating *315, 342*
  procedures *355*
  scan procedures for *156*
  searching *324, 327*
directory
  current *330, 331*
  remove a *352, 355*
DISCARDABLE code segment attribute *192*
DiskFree function *158, 317*
disks
  status functions *158*
disks, space *317*
DiskSize function *158, 317*
Dispose procedure *142, 317*
  extended syntax *207, 215*
    constructor passed as parameter *106, 215*
div operator *69*
DLL
  CmdShow variable *137*
  exit code *135*
  global variables in *137*
  HeapLimit variable *138*
  HPrevInst variable *137*
  initalization code *135*
  PrefixSeg variable *137*
  run-time errors in a *138*
  structure of *132*
  unloading a *136*
  using a *128*
  using files in a *137*
  using global memory in a *137*
  writing a *132*
DMT cache *211*
DMT entry count *211*
domain, object *33*
DoneWinCrt procedure *172, 176, 177, 318*
DOS
  device handling *234*
  devices *150*
  error codes *157*
  error level *232*
  exit code *231*
  operating system routines *153*
  Pascal functions for *343*
  registers and *156*
  verify flag *335*
    setting *362*
DosError variable *157, 327, 328, 332, 333, 359, 360*
DosVersion function *160, 318*
Double floating-point type *183, 199*
DS register *231, 233, 292, 299, 319*
  value of *319*
DSEG *291*
DSeg function *144, 319*

DX register *227, 233*
dymamic method table *209*
dynamic
  memory allocation *141*
    functions *141*
  variables *41, 48, 53*
dynamic importing *131*
dynamic-link libraries *127*
dynamic method calls *213*
dynamic method table
  cache *211*
  entry count *211*
dynamic methods *209*
dynamic object instances
  allocation and disposal *106, 215*

E

$ELSE compiler directive *261*
empty set *40*
emulating the 80x87 *182*
emulation the 80x87 *187*
end of file
  error messages *389*
end of file status *320*
end-of-file status *319, 357*
end-of-line status *320, 357*
$ENDIF compiler directive *261*
entry code, procedures and functions *229*
enumerated type *25, 197*
EnvCount function *159*
environment variable *332*
EnvStr function *159*
Eof function *146, 319, 320*
Eoln function *148, 320*
Erase procedure *146, 321*
error checking
  dynamic object allocation *220*
  virtual method calls *207*
ErrorAddr variable *151, 232*
errors
  messages *387*
    fatal *409*
  range *251*
  reporting *231*
  run-time *See* run-time, errors
ES register *233*
.EXE files
  building *242*
exit
  functions *229*
  procedures *229, 231, 322*
Exit procedure *141*
ExitCode variable *135, 151, 232*
exiting a program *231*
ExitProc variable *135, 231*
Exp function *143, 322*
exponents *198*
export directive *133*
exports clause *133*
expressions *65*
  constant *12*
    address *57*
  examples *68*
  inline assembler *274*
    classes *281-282*
    elements of *275-281*
Extended
  floating-point type *183, 199*
  range arithmetic *183*
extended
  syntax *162, 253*
Extended Syntax option *253*
extensibility *39*
external
  declarations *101, 255, 291*
  procedure errors *393*
external (reserved word) *217*
external declaration *128*
EXTRN definition errors *292, 393*

F

$F compiler directive *228, 249*
factor (syntax) *66*
Fail procedure *221*
far
  directives *99*
far calls *227*
  model
    forcing use of *231, 249*
fatal run-time errors *409*
Fibonacci numbers *186*
field
  designators *52*

list (of records) *31*
record *52*
fields, object *33*
designators *53*
scope *36, 105*
file
size of *325*
split into components *325*
file componet
read *351*
file-handling functions *159*
file-handling procedures *158*
Rename *352*
Reset *353*
Rewrite *354*
Seek *356*
SetFAttr *359*
Truncate *378*
file name
component constants *155*
expanding *323*
file position *323*
FileExpand function *155, 159, 323*
FileMode variable *149, 151, 152*
FilePos function *146, 323*
files
access, read-only *149*
access-denied error *407*
.ASM *187*
Assign procedure *308*
attributes *332*
constants *154*
buffer *202*
closing *313*
erasing *321*
.EXE
building *242*
functions for *146*
handles *201*
I/O *141*
modes *201*
constants *154*
.OBJ *291*
linking with *255*
procedures for *146*
record types *155*
text *147*
typed *152, 201*
types *40, 201*
untyped *148, 152, 201*
untyped, variable *309, 311*
FileSearch function *155, 159, 324*
FileSize function *146, 325*
FileSplit function *155, 159, 323, 325*
return flags for *155*
FillChar procedure *145, 326*
FindFirst procedure *154, 158, 327*
TSearchRec and *156*
FindNext procedure *154, 158, 328*
TSearchRec and *156*
FIXED code segment attribute *191*
fixed part (of records) *31*
flags constants *153*
floating-point
calculations, type Real and *183*
code generation, switching *182*
errors *410*
numbers *181*
numeric coprocessor (80x87) *27*
parameters *226*
routines *141*
software *27*
types *See* types, floating-point
Flush function *237*
Flush procedure *148, 328*
FlushFunc function *236*
for statements, syntax *92*
Force Far Calls
command *249*
option *249*
force far calls compiler switch *249*
formal parameters *85, 108*
forward declarations *100*
Frac function *143, 329*
fractions, returning *329*
free bytes on disk
number of *317*
FreeMem procedure *142, 329*
functions *97*
address *144*
arithmetic *143*
body *103*
calls *77, 225*
declarations *103*

disk status *158*
dynamic allocation *142*
entry/exit code
  inline assembler *288*
extended syntax *78, 253*
file-handling *159*
headings *103*
heap error *220*
importing *128*
inline assembler *287-290*
methods denoting *78*
nested *113*
ordinal *143*
parameters
  inline assembler *287*
pointer *144*
results *227*
  discarding *78, 253*
returns
  inline assembler *288*
SizeOf *211*
stack frame
  inline assembler *288*
standard *141*
string *144*
transfer *142*
TypeOf *212*

G

$G compiler directive *249*
GetArgCount function *330*
GetArgStr function *330*
GetCBreak procedure *159, 330*
GetCurDir function *330*
GetCurDir procedure *159*
GetDate procedure *158, 331*
GetDir procedure *146, 331*
GetEnv function *159*
GetEnvVar function *332*
GetFAttr procedure *154, 158, 332*
GetFTime procedure *158, 333*
GetIntVec procedure *158, 334*
GetMem procedure *142, 334*
GetTime procedure *158, 334*
GetVerify procedure *159, 335*
global heap *194*
global variables
  in a DLL *137*
goto statements *86*
GotoXY procedure *176, 178, 335*

H

Halt procedure *141, 231, 335*
handles
  file *201*
handles, DOS *308*
hardware, interrupts *233*
heap
  free memory in *342*
heap error function *220*
heap management *194*
  fragmenting *194*
  sizes *256*
heap manager *194*
  allocating memory blocks *195*
HeapBlock variable *151, 195*
HeapError variable *151, 195*
HeapLimit variable *151, 195*
HeapList variable *151*
hexadecimal constants *9*
Hi function *144, 239, 336*
high
  order bytes *336*
HInstance variable *137, 151*
host type *26*
HPrevInst variable *151*

I

$I compiler directive *146, 250, 255, 339*
I/O *145*
  checking *250, 339*
  devices *235*
  DOS standard *308*
  error-checking *146, 250*
  errors *408*
  files *141*
    standard *152*
I/O Checking
  command *250*
  option *250*
identifiers *7*
if statements *88*

$IFDEF compiler directive *260*
$IFNDEF compiler directive *261*
$IFOPT compiler directive *261*
implementation
  constructors *105*
  destructors *105*
  methods *36, 105*
  part (program) *120, 228*
  sections *124*
import units *129*
importing procedures and functions *128, 129*
  dynamically *131*
  statically *131*
in operator *73, 75*
Inc procedure *143, 336*
include directories command-line option *255*
Include files *255*
  nesting *255*
including resources *257*
index clause *128, 134*
index expressions *51*
indexing
  character pointers *164*
indirect unit references *121*
inheritance *33*
initialization part (program) *121*
initialized variables *57*
InitWinCrt procedure *172, 176, 177, 337*
inline
  declarations *102*
  directives *300*
  machine code *298*
  statements *298*
inline assembler
  asm statement *266*
  assembler directive *287*
  comments *266, 267*
  constants *275-277*
    numeric *275*
    string *276-277*
    untyped *281*
  directives *265, 271-273*
    assembler
      external versus *288*
  expressions *274-287*
    classes *281-282*
    elements of *275-281*
    immediate values *281*
    operators *284-287*
    Pascal expressions versus *274*
    registers *281*
    types *282-284*
  labels *267-269*
  memory references *281*
  opcodes
    instruction *269-271*
      sizing *270-271*
    prefix *269*
  operands *273-274*
  operator precedence *284*
  procedures and functions *287-290*
  registers *277*
    use *266*
  relocation *282*
  reserved words *273*
  separators *266*
  symbols *278-281*
    invalid *278*
    scope access *280*
    special *278*
  syntax *267*
InOut function *237*
InOutRes variable *151, 152*
input
  files *152*
input, DOS standard *308*
Input file *147*
Input standard file *152*
Input text file
  WinCrt unit in *171*
Input variable *151*
Insert procedure *143, 337*
inserting
  strings *337*
instances
  dynamic objects *38*
  object *37*
Int function *143, 337*
Integer data type *23, 197*
interface section (program) *119, 125, 228, 292*
interfacing Turbo Pascal with Turbo Assembler *292*
internal data formats *197*

interrupt
  directives *100*
  handlers *233*
  handling routines *233*
  procedures *360*
  service routines (ISRs) *233*
  support procedures *158*
  vectors *334*
Intr procedure *158, 338*
  registers and *156*
invalid typecasting errors *395*
IOResult function *146, 152, 339*
IP flag *233*
ISRs (interrupt service routines) *233*

K

keyboard
  read character from *351*
  scan codes *416*
KeyPressed function *173, 177, 178, 339*

L

$L compiler directive *217, 250, 255, 291*
labels *9*
  declaration part *16*
  local *268*
late binding *36*
Length function *144, 239, 340*
libraries
  dynamic-link *127*
library header *132*
linking
  assembly language *291*
  object files *255*
  smart *242*
Ln function *143, 340*
Lo function *144, 239, 340*
local heap *193, 194*
Local labels *267*
local symbol information switch *250*
Local Symbols
  command *250*
  option *250*
logarithm
  natural *340*
logical operators *70*
LongBool type *24, 197*
Longint data type *23*
low-order byte *340*

M

$M compiler directive *48, 256, 341, 342*
machine code *298*
macros, inline *300*
Mark procedure *194*
math coprocessor *See* numeric coprocessor
MaxAvail function *142, 341*
Mem array *202*
MemAvail function *142, 342*
MemL array *202*
memory *329, 334*
  allocation
    compiler directive *256*
  error messages *387*
  inline assembler references *281*
  size *256*
Memory Sizes command *256*
memory usage
  Turbo Pascal and *191*
MemW array *202*
methods
  activation, qualified *85*
  assembly language *217*
  calling
    as functions or procedures *78, 85*
    conventions *85, 214*
    dynamic *213*
  declaring *105*
  defined *33*
  designators *39, 53*
    @ (address operator) with *77*
  dynamic *209, 213*
  external *217*
  identifiers, qualified *36*
    accessing object fields *53*
    in method calls *39, 85*
    in method declarations *105*
  implementation *36, 105*
  overridden, calling *86*
  overriding inherited *37*
  parameters
    Self *85, 86, 105*
      defined *214*

type compatibility *110*
qualified activation *85*
static *36*
calling *39*
virtual *36*
calling *39, 85, 212*
error checking *207*
methods. declaring *36*
MkDir procedure *146, 342*
mod operator *70*
modular programming *118*
Move procedure *145, 343*
MOVEABLE code segment attribute *191*
moving the cursor *335*
MsDos procedure *158, 343*

N

$N compiler directive *27, 182, 187, 256*
name clause *128, 134*
near
directives *99*
NEAR calls *227*
nesting
files *255*
procedures and functions *228*
network file access, read-only *149*
New procedure *41, 142, 344*
extended syntax *207*
constructor passed as parameter *106, 215, 344*
used as function *216*
nil *41, 53*
not operator *71*
NULL character *161*
null strings *11, 28*
null-terminated strings *41, 161*
NULL character *161*
pointers and *162*
standard procedures and *167*
number
random *348*
numbers, counting *9, 197*
numeric coprocessor
compiler switch *256*
detecting *187*
emulating *141*
assembly language and *187*
evaluation stack *185*
floating-point *27*
mode *400*
numeric processing option *27*
using *181-187*

O

.OBJ files *291*
linking with *255*
object
directories, compiler directive *255*
files *291*
linking with *255*
segment of *358*
objects
ancestor *33*
constructors
declaring *105*
defined *106*
error recovery *220*
implementation *105*
inherited *37*
virtual methods and *38, 106*
VMTP and *205, 208*
descendant *33*
destructors *106*
declaring *105*
defined *106*
implementation *105*
domain *33*
dynamic instances *38*
allocation and disposal *106, 215*
dynamic method table *209*
fields *33*
designators *53*
private
scope *36*
scope *36, 105*
inheritance *33*
instances *37*
internal data format *205*
methods
private
scope *36*
pointers to *39*
polymorphic *39, 84, 110*
typed constants of type *62*

types *33*
declaring *34*
virtual method table *207*
pointer *205*
initialization *208*
virtual methods
call error checking *207*
calling *212*
Odd function *143, 239, 344*
odd number *344*
offset of an object *345*
Ofs function *144, 345*
opcodes *298*
inline assembler *269-271*
Open function *236*
operands *65*
operations
character pointer *166*
operators *6, 65, 69*
@ (address-of) *41, 53*
address-of (@) *81*
and *71*
arithmetic *69*
bitwise *70*
Boolean *70*
div *69*
logical *70*
mod *70*
not *71*
or *71*
PChar *72*
precedence
inline assembler *284*
precedence of *65, 69*
relational *73*
set *73*
shl *70*
shr *70*
string *72*
xor *71*
optimization of code *239*
or operator *71*
Ord function *23, 25, 142, 239, 345*
order of evaluation *240*
ordinal
functions *143*
procedures *143*
types *22*
ordinal number of value *345*
out-of-memory errors *387*
output
files *152*
output, DOS standard *308*
Output file *147*
Output standard file *152*
Output text file
WinCrt unit in *171*
Output variable *151*
overlays
manager *141*
overridden methods, calling *86*
overriding inherited methods *37*

P

Pack procedure *142*
packed (reserved word) *29*
PackTime procedure *158, 345*
TDateTime and *156*
ParamCount function *144, 346*
parameter
command-line *330*
parameter directives *253, See* compiler, directives, parameter
parameters
actual *85*
command-line *144*
floating-point *226*
formal *85, 108*
number of *346*
number passed *330*
passing *85, 225*
procedural-type *114*
Self *85, 86, 105*
defined *214*
type compatibility *110*
value *109, 226*
variable *109*
untyped *110*
VMT *214*
parameters, command-line *346*
ParamStr function *144, 346*
Pascal strings *161*
PChar operators *72*
PChar type *41, 162*

PERMANENT code segment attribute *192*
Pi function *143, 347*
pointer (^) symbol *41, 53*
pointer and address functions *144*
pointer-type constants *62*
pointers
  assignment compatibility *39*
  comparing *74*
  comparing character *75*
  to objects *39*
  types *41, 200*
  values *53*
  variables *53, 76*
polymorphism
  object instance assighment *84*
  parameter type compatibility *110*
  pointer assignment *39*
Port array *237*
PortW array *237*
Pos function *144, 347*
pound (#) character *11*
precedence of operators *65, 69*
Pred function *23, 143, 239, 347*
PrefixSeg variable *151*
PRELOAD code segment attribute *192*
private
  fields
    scope *36*
  methods
    scope *36*
PROC directive, defining parameters with *293*
procedural
  types *42, 111, 111-115*
    declarations *42*
    in expressions *79*
    in statements *79*
    variable declaration *111*
    variable typecasts and *54*
  values, assigning *111*
  variables *111*
    restrictions *112*
    using standard procedures and functions with *112*
procedural-type constants *63*
procedural-type parameters *114*
procedure and function declaration part (program) *17*
procedure call models *99*
procedures *97*
  body *98*
  declarations *97*
  Dispose
    extended syntax *207, 215*
      constructor passed as parameter *106, 215*
  dynamic allocation *141*
  entry/exit code
    inline assembler *288*
  Exit *141*
  Fail *221*
  file-handling *158*
  Halt *141*
  headings *98*
  importing *128*
  inline assembler *287-290*
  methods denoting
    calls to *85*
  nesting *113, 228*
  New
    extended syntax *207*
      constructor passed as parameter *106, 215*
    used as function *216*
  ordinal *143*
  parameters
    inline assembler *287*
  stack frame
    inline assembler *288*
  standard *141*
  statements *85*
  string *143*
programs
  execution, stopping *356*
  halting *335*
  headings *117*
  lines *14*
  parameters *117*
  syntax *117*
  termination *231*
Ptr function *41, 144, 239, 348*
PUBLIC *291*
  definition errors *393*

Q

qualified
  activation *85, 86*
  identifiers *8, 18*
  method identifiers *36*
    accessing object fields *53*
    in method calls *39, 85*
    in method declarations *105*

R

$R compiler directive *251*
  virtual method checking *207*
$R filename directive *257*
Random function *144, 152, 348*
random generator
  initialize *349*
random number *348*
random number generator *152*
Randomize procedure *145, 349*
RandSeed function *152*
RandSeed variable *151*
Range Checking
  command *251*
  option *251*
range checking *165*
  compile time *241*
  compiler switch *251*
range checking, Val and *379*
read a line *349, 352*
read file component *351*
read keyboard character *351*
read-only file access *149*
Read procedure
  text files *145, 148, 349*
  typed files *351*
read text file *349*
ReadBuf function *177, 178*
reading records *309*
ReadKey function *173, 177, 178, 351*
Readln procedure *148, 352*
real
  numbers *27, 181, 198*
  types *27*
record-type constants *61*
records *30, 52, 61, 201*
  fields *52*
redeclaration *17, 47*
reentrant code *233, 234*
referencing errors *400*
register-saving conventions *231*
registers
  AX *227, 300*
  BP *231, 233, 299*
  BX *227, 233*
  CS *233*
  CX *233*
  DI *233*
  DS *231, 233, 292, 299*
  DX *227, 233*
  ES *233*
  inline assembler *281*
  inline assembler use *266*
  SI *233*
  SP *231*
  SS *231*
  using *227, 231, 233, 299*
registers, CS *315*
relational operators *73*
relaxed string parameter checking *252*
Release procedure *194*
relocatable reference errors *394*
relocation
  inline assembler *282*
RemoveDir procedure *159, 352*
Rename procedure *146, 352*
repeat statements *90*
repetitive statements *90*
reserved words *6, 7*
  external *217*
  inline assembler *273*
  virtual *36*
  with *53*
Reset procedure *145, 146, 152, 353*
resident keyword *134*
resource file *257*
resources
  including *257*
results
  functions
    discarding *78, 253*
Rewrite procedure *145, 146, 354*
RmDir procedure *146, 355*
Round function *142, 239, 355*

round-off errors, minimizing *184*
rules, scope *17*
run-time
  errors *231, 406*
    fatal *409*
    in a DLL *138*
  support routines *141*
run-time errors, generating *356*
RunError procedure *356*

S

$S compiler directive *48, 251*
scale factor *10*
scan codes, keyboard *416*
scope
  object *35*
  of declaration *17*
ScreenSize variable *172*
ScrollTo procedure *176, 179*
searching directories *327*
Seek procedure *145, 147, 356*
SeekEof function *148, 357*
SeekEoln function *148, 357*
Seg function *358*
segment attributes *191*
segment of object *358*
Self parameter *85, 86, 105*
  defined *214*
Seq function *144*
set-type constants *62*
set types *40, 200*
SetCBreak procedure *159, 358*
SetCurDir procedure *159, 358*
SetDate procedure *158, 359*
SetFAttr procedure *154, 159, 359*
SetFTime procedure *158, 360*
SetIntVec procedure *158, 360*
sets
  comparing *75*
  constructors *66, 78*
  membership *75*
  operators *73*
SetTextBuf procedure *148, 360*
SetTime procedure *158, 362*
SetVerify procedure *159, 362*
shl operator *70*
short-circuit Boolean evaluation *240, 248*
Shortint data type *23*
shr operator *70*
SI register *233*
signed number (syntax) *10*
significand *198*
simple
  statements *83*
  types *22*
simple-type constants *58*
Sin function *143, 362*
sine of argument *362*
Single floating-point type *183, 198*
size
  disk *317*
  free memory
    in heap *342*
  largest free block
    in heap *341*
  of argument *363*
SizeOf function *144, 211, 327, 363*
smart linking *242*
software
  floating-point *27*
  interrupts *233*
  numeric processing *See* numeric coprocessor, emulating
software interrupts *338*
source debugging compiler switch *248*
SP register *231, 299*
  value of *363*
space characters *5*
SPtr function *144, 363*
Sqr function *143, 364*
Sqrt function *143, 364*
square
  of argument *364*
square root
  of argument *364*
SS register *231, 299*
  value of *364*
SSeg function *144, 364*
stack *193*
  80x87 *185*
  checking switch directive *251*
  overflow *48*
    switch directive *251*
  segment *48*

size *256*
Stack Checking
command *251*
option *251*
stack frame
inline assembler use of *288*
stack segments
DLLs and *139*
standard
functions, constant expressions and *13*
units *See* units, standard
statement part (program) *17*
statements *83*
assignment *84*
case *89*
compound *87*
conditional *88*
for *92*
goto *86*
if *88*
procedure *85*
repeat *90*
repetitive *90*
simple *83*
structured *87*
uses *118*
while *91*
with *53, 94*
static data area *193*
static importing *131*
static methods *36*
calling *39*
Str procedure *143, 364*
StrCat function *167, 365*
StrComp function *167, 366*
StrCopy function *167, 366*
StrDispose function *168, 367*
StrECopy function *167, 367*
StrEnd function *167, 368*
StrIComp function *168, 368*
strict string parameter checking *252*
string
length
determining *340*
string literal
assigning to PChar *162*
string-type constants *59*
String Var Checking
command *252*
option *252*
strings *59*
allocating on heap *372*
appending *365, 368*
character *11*
comparing *74, 366, 369, 371*
concatenation *72, 314*
construction *314*
converting *373*
converting to lowercase *371*
converting to uppercase *376*
copying *366, 367, 373*
copying characters *370, 372*
coverting *161*
deletion *316*
first occurrence of character *375*
first occurrence of string *374*
functions *144*
handling *141*
initializing *326*
last occurrence of character *375*
length *370*
length byte *200, 326*
maximum length *200*
null *28*
null-terminated *41, 161*
operators *72*
Pascal *161*
procedures *143, 337, 364, 379*
relaxed parameter checking *252*
strict parameter checking *252*
types *28, 200*
variables *51*
Strings unit *161*
using the *167*
StrLCat function *167, 368*
StrLComp function *168, 369*
StrLCopy function *167, 370*
StrLen function *167, 370*
StrLIComp function *168, 371*
StrLower function *168, 371*
StrMove function *167, 372*
StrNew function *168, 372*
StrPas function *168, 373*
StrPCopy function *167, 373*

StrPos function *168, 374*
StrRScan function *168, 375*
StrScan function *168, 375*
structured
  statements *87*
  types *28*
    declaring *113*
structured-type constants *59*
StrUpper function *168, 376*
subdirectory
  create a *342*
subrange type *26*
substrings
  copying *314*
  deleting *316*
  inserting *337*
  position *347*
Succ function *23, 143, 239, 376*
Swap function *144, 239, 377*
swapping bytes *377*
switch compiler directives *245, 246-252*
switch directives *246*
symbols *5*
  conditional *258*
  CPU *260*
  inline assembler *278-281*
  local information *250*
  scope access
    inline assembler *280*
syntax
  extended *253*
  inline assembler *267*
syntax diagrams, reading *5*
System unit *118, 141, 187*
  floating-point routines *182*

T

tag field (of records) *32*
task header *193*
TDateTime type *156*
terminating a program *231, 322*
terms (syntax) *67*
text
  files *147, 319*
    devices *150*
      drivers *235*
    records *202*
text file
  read from *349*
TFileRec *155, 201*
time procedures
  GetFTime *333*
  GetTime *334*
  SetFTime *360*
  SetTime *362*
tokens *5*
TrackCursor procedure *176, 179, 377*
transfer functions *142*
trapping
  I/O errors *250*
  interrupts *233*
TRegisters type *156*
Trunc function *142, 239, 377*
truncate a value *377*
Truncate procedure *147, 378*
TSearchRec type *156*
TTextRec records *155, 201, 235*
Turbo Assembler *291, 292*
  80x87 emulation and *187*
  example program *297*
type
  Boolean *197*
  LongBool *197*
  WordBool *197*
type checking, strings and *252*
typecasting, invalid *395*
typed
  constants *57*
    object type *62*
  files *152, 201*
TypeOf function *212*
types *21*
  array *29, 200*
  Boolean *24*
  Byte *23*
  Char *25, 197*
  common *24*
  compatibility *44*
  declaration *21*
    part *16, 45*
  definition, constant expressions and *26*
  enumerated *25, 197*
  file *40*
  floating-point *183, 198*

Comp *27, 183, 199*
comparing values of *185*
Double *27, 183, 199*
Extended *27, 183, 199*
Single *27, 183, 198*
host *26*
identity *43*
Integer *23, 197*
LongBool *24*
Longint *23*
mismatches, error messages *390*
object *33*
declaring *34*
ordinal *22*
PChar *41, 162*
Pointer *41, 200*
procedural *42, 79, 111*
real *27*
real numbers *198*
record *30, 201*
set *40, 200*
Shortint *23*
simple *22*
string *28, 200*
structured *28*
subrange *26*
Word *23*
WordBool *24*

U

$UNDEF compiler directive *258, 260*
unit
WinCrt *147, 171*
units
80x87 coprocessor and *187*
circular references *122*
heading *119*
identifiers *7*
indirect references *121*
scope of *18*
standard
System *141*
system *118*
WinCrt *150*
WinDos *153*
syntax *118*
version
mismatch errors *396*
number *122*
Unpack procedure *142*
UnpackTime procedure *158, 378*
TDateTime and *156*
unsigned
constant *66*
integer *9*
number *10*
real *10*
untyped
files *148, 152, 201*
var parameters *110*
untyped files, variable *309, 311*
UpCase function *144, 378*
uses statement *118*

V

$V compiler directive *252*
Val procedure *143, 379*
value
parameters *109, 226*
typecasts *79*
var
declaration section *243*
parameters *109, 226*
untyped *110*
string checking, compiler switch *252*
variable
decrementing a *316*
dynamic
create *344*
environment *332*
variables *47*
absolute *49*
arrays *51*
declaration part *16*
declarations *47*
disposing of *317, 329*
DosError *157, 327, 328, 332, 333, 359, 360*
dynamic *41, 53*
FileMode *149*
global *48*
increasing *336*
initializing *57*
local *48*

parameters *226*
pointer *53, 76*
procedural *111*
restrictions *112*
record *52*
references *50*
strings *51*
typecasts *53, 54*
untyped file *309, 311*
variant part (syntax) *31*
VER10 symbol *259*
video operations
Write (text) *380*
Write (typed) *382*
Writeln *383*
virtual (reserved word) *36*
virtual method table *207*
pointer *205*
initialization *208*
virtual methods *36*
calling *39, 85, 212*
error checking *207*
VMT field *205*
VMT parameter *214*

W

$W compiler directive *252*
WEP
exported function *136*
WhereX function *177, 179*
WhereX procedure *380*
WhereY function *177, 179*
WhereY procedure *380*
while statements (syntax) *91*
WIN87EM.DLL emulator library *182, 187*
WinCrt unit *147, 150, 171*
using the *171*
variables in *174*
WinDos unit *153*
constants *153*
date and time procedures *158*
disk status functions *158*
DosError in *157*
environment-handling functions *159*
file-handling procedures and functions *158, 159*
interrupt support procedures *158*
miscellaneous functions *160*
miscellaneous procedures *159*
types *155*
windows stack frames *252*
WINDOWS symbol *259*
with (reserved word)
statement *53*
with statements *94*
Word
data type *23*
word
alignment
automatic *241*
WordBool type *24, 197*
write
procedures *145*
statements
80x87 coprocessor and *186*
Write procedure *148*
text files *380*
typed files *382*
write statements, DOS *308*
WriteBuf function *383*
WriteBuf procedure *176, 177*
WriteChar procedure *177, 178*
Writeln procedure *148, 383*
80x87 coprocessor and *186*
writing records *311*

X

$X compiler directive *162, 253*
xor operator *71*

Z

zero-based character arrays *60, 162, 163*

PROGRAMMER'S GUIDE

TURBO PASCAL®
FOR WINDOWS

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom ■ PART # 11MN-TPW03-10 ■ BOR 2109